



游戏编程精粹4

GAME PROGRAMMING

Gems 4

[美] Andrew Kirmse 编
沙鹰 等译



dearbook.com

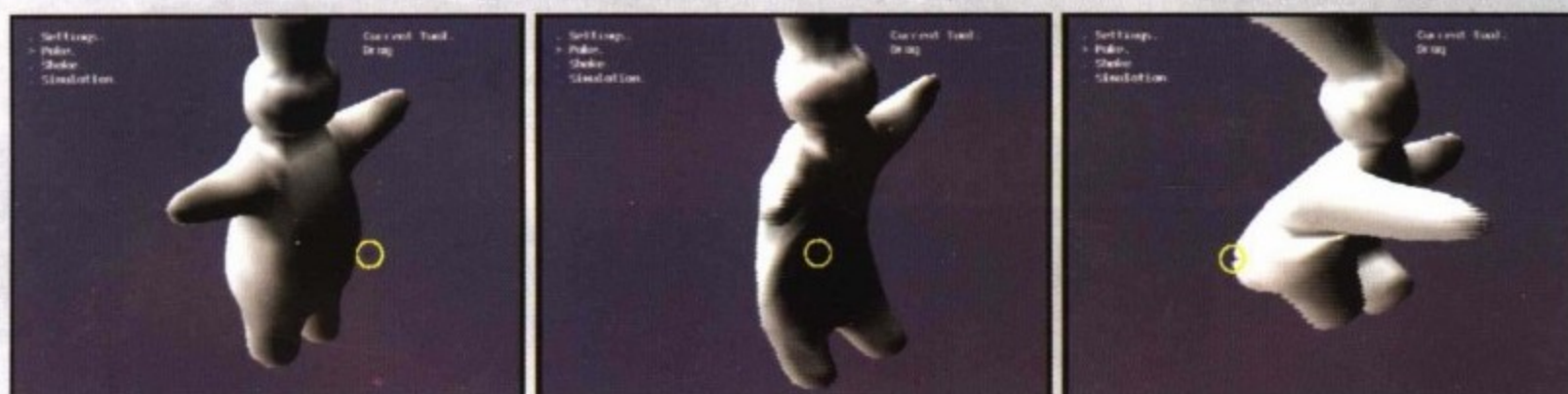
人民邮电出版社
POSTS & TELECOM PRESS



彩图 1

这是从两个物体相撞的动画中抽取出来的6帧图。两个物体各由一个刚体部分和一个可发生形变的部分混合模拟而成。感谢 J.O' Brien、C.Shen 和 K.Hauser 提供图片。

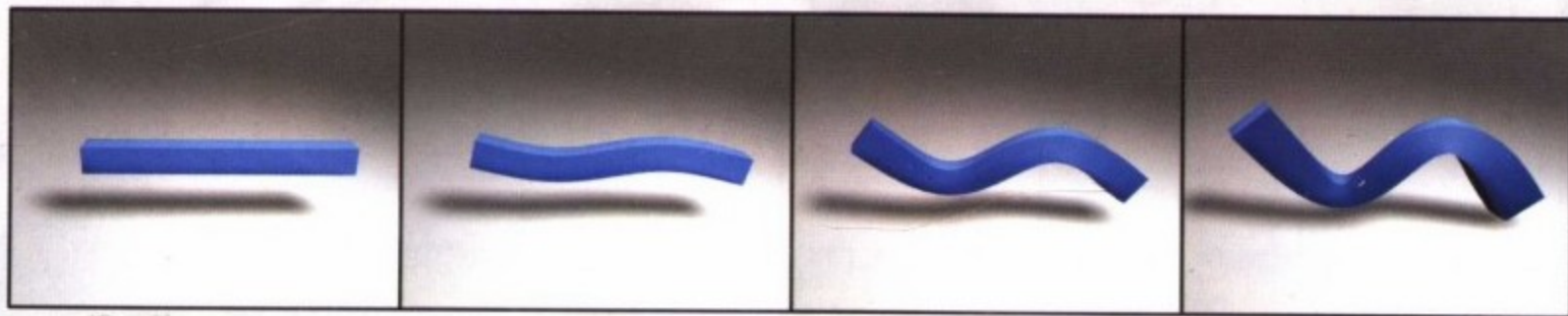
2003 加州大学伯克利分校版权所有。使用已获得许可。



彩图 2

这几张图是在 Sony PlayStation 2 游戏机上运行的演示程序的屏幕拷贝。黄色圆圈标出了用户正对这个塑胶兔子指指戳戳的光标位置。感谢 J.O' Brien、C.Shen 和 K.Hauser 提供图片。

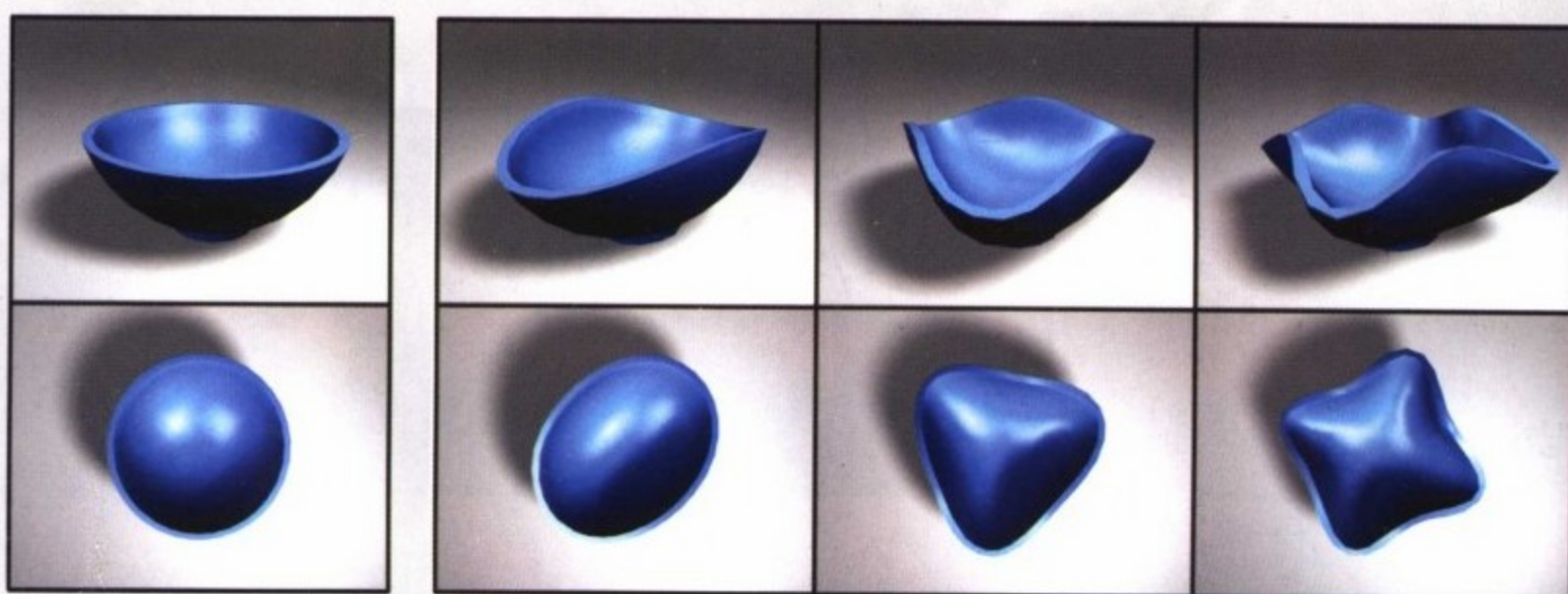
2003 加州大学伯克利分校版权所有。使用已获得许可。



彩图 3

本图展示了线性化操作带来的失真。第1张图是一根发生形变之前的棒。接下来的图显示了逐渐增大形变的结果。轻度到中度的形变带来的误差不容易用肉眼察觉；但高度形变带来易被察觉的失真，如最后一张图中整条棒上均有易被察觉的失真。感谢 J.O' Brien、C.Shen 和 K.Hauser 提供图片。

2003 加州大学伯克利分校版权所有。使用已获得许可。



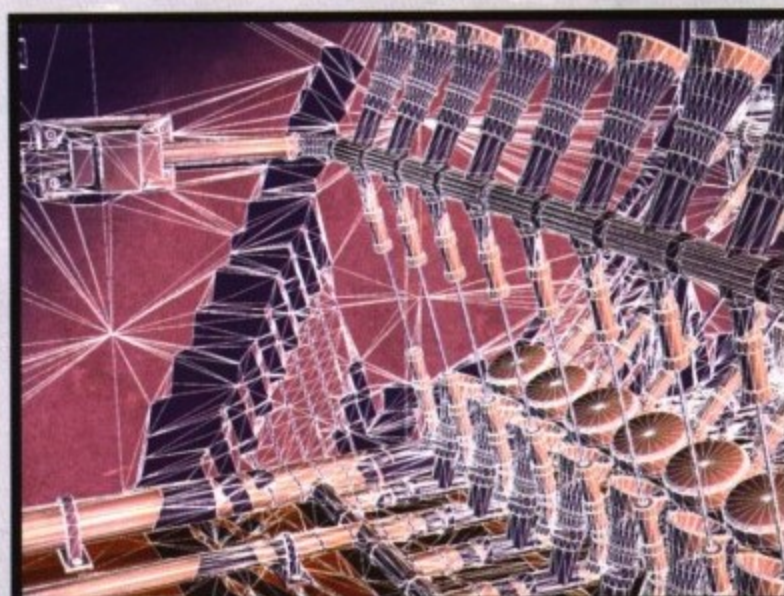
彩图 4

第1行是侧视图，第2行是顶视图。从左到右分别是碗的初始状态和按特征值排序的3个振动模式。图中表现的模式是前3个非刚体模式，具有不同的特征值，受沿碗口方向的一个横向脉冲激励。

感谢 J.O' Brien、C.Shen 和 K.Hauser 提供图片。

2003 加州大学伯克利分校版权所有。使用已获得许可。

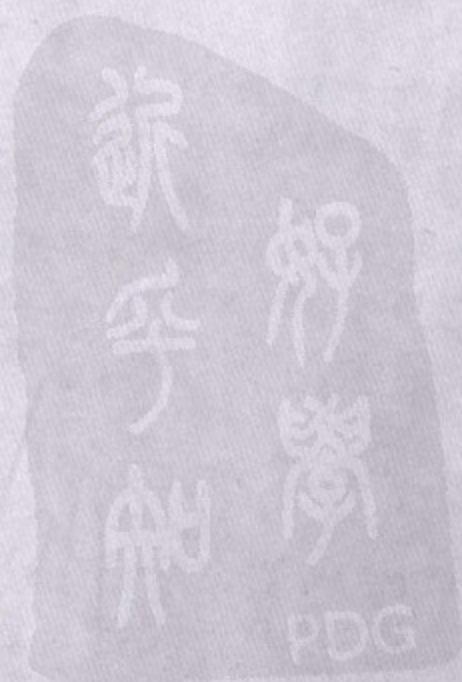
数字图书馆
PDG

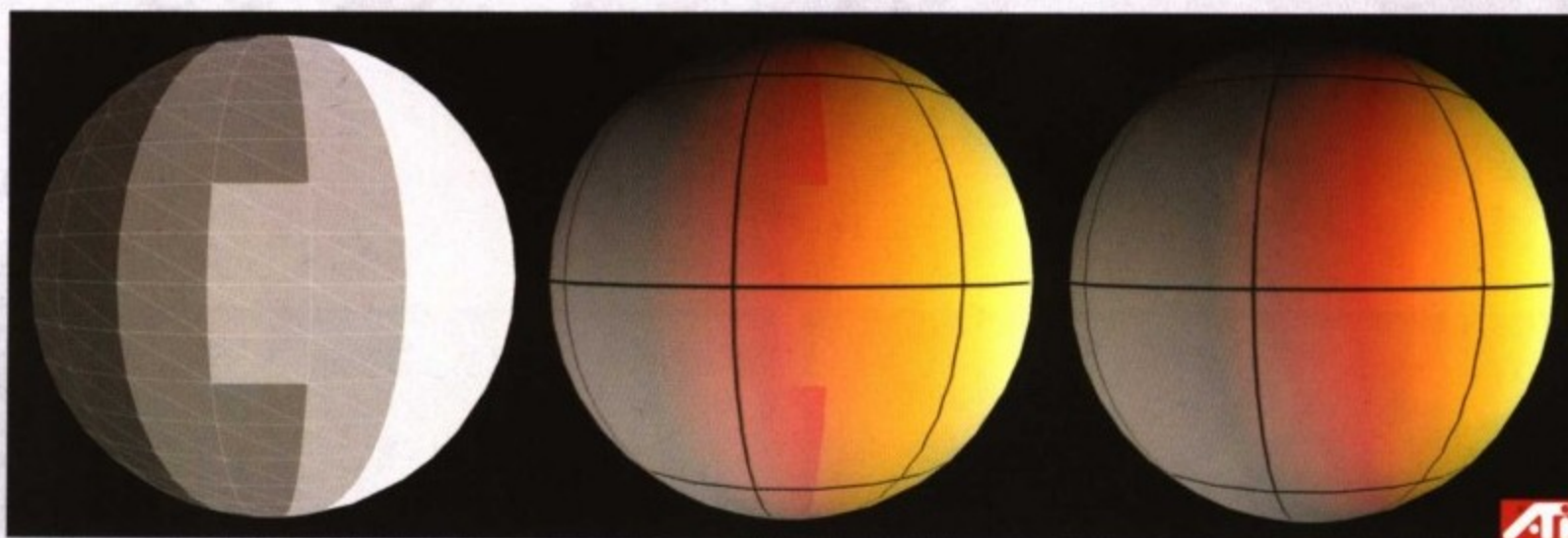


彩插 3

白色线框表示了快速绘制静态阴影所用的模型的细分。

2003 ATI Technologies, Inc. 版权所有。使用已获得许可。





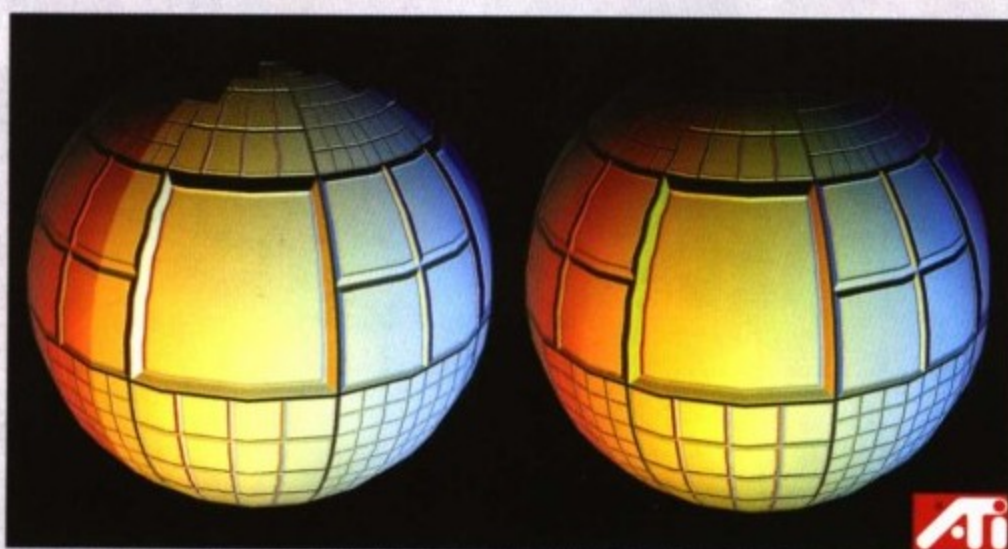
彩图 6

[左图]根据场景中照射到球体的面的光源个数，而将球体划分成数个网格模型。最亮的模型受到3盏光源的照耀，最暗的模型则不受任何光源的直接照射。

[中图]之后，每个模型都用一个与参与照射的光源的个数相对应的shader进行照明。光源参与照射的程度在预处理阶段通过面法向量来确定，因为在运行时用顶点法向量计算照明度会造成亮度不连续。

[右图]调整漫射光线，使光线柔和地变暗，以补偿模型划分带来的差异。

2003 ATI Technologies, Inc. 版权所有。

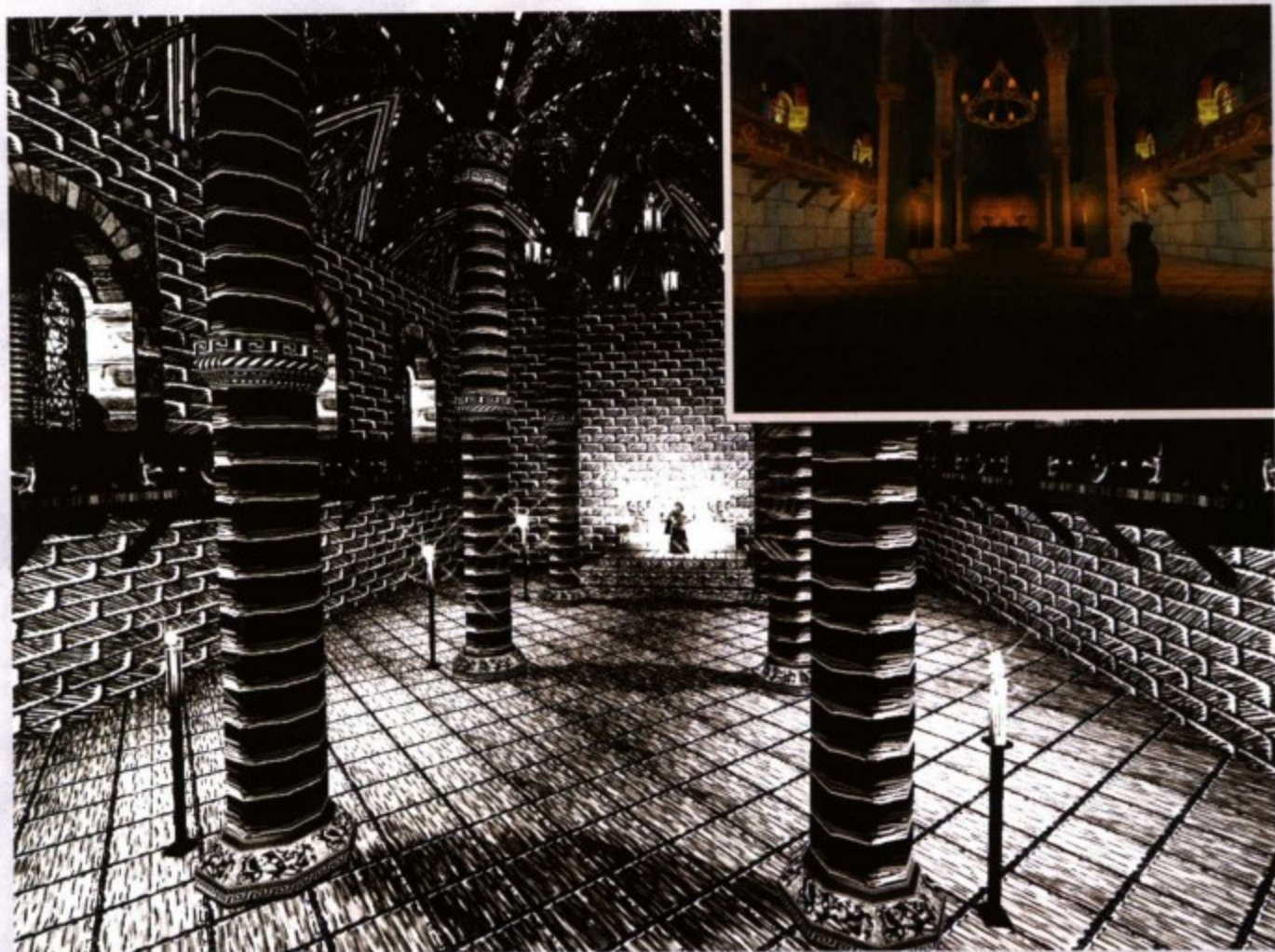


彩图 7

[左图]以逐像素计算照射度的方式，显示了一个采用凹凸贴图的球体，该球体已经根据照射到表面的静态场景光源个数，而被划分成数个网格模型。在顶点法向量与面法向量相差太多的地方，我们可以看到照明的不连续。

[右图]调整漫射光线，以补偿照明的不连续性。

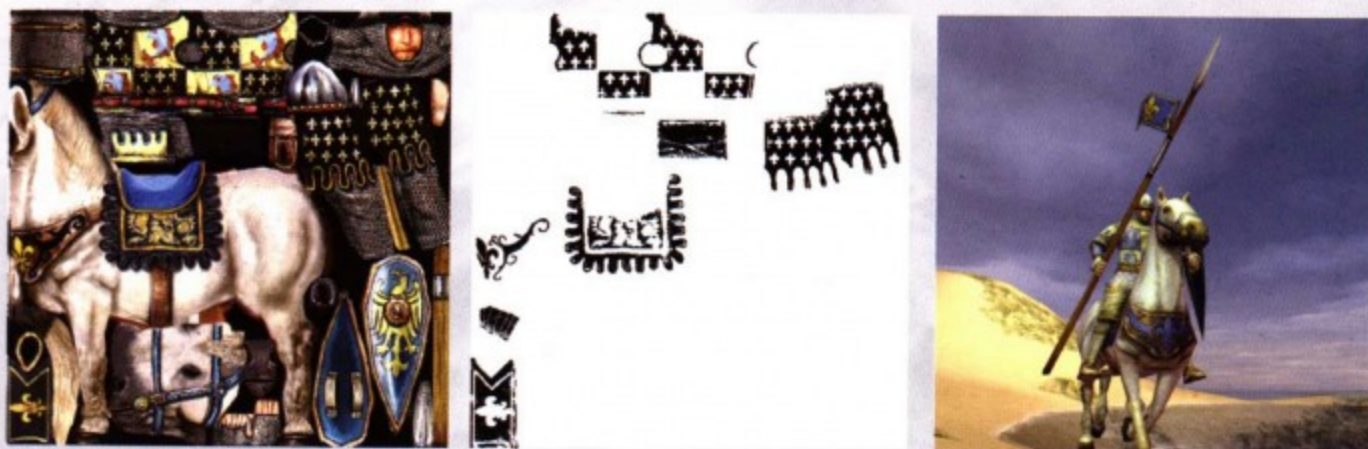
2003 ATI Technologies, Inc. 版权所有。



彩插 8

通过实时半色调法，将一个普通的使用了贴图 and 照明的场景（如右上角小图所示）画面转换成黑白漫画风格。留意其中是怎样采用多重笔触来表现明暗的。

2001 Bert Freudenberg, isg. 版权所有。使用已获得许可。



彩图 9

[左图]用于士兵单位身上的贴图和 alpha 图。Alpha 图指出了应着团队色的场合。

[中图]红色单位。

[右图]采用同样贴图的蓝色单位。

2003 Stainless Steel Studios, Inc. 版权所有。Empires: Dawn of the Modern World 是 Stainless Steel Studios, Inc. 的商标。使用已获得许可。



彩图 10

实时地将一幅彩色画面转换成棕褐色色调。感谢 Marwan Y. Ansari 提供图片。使用已获得许可。

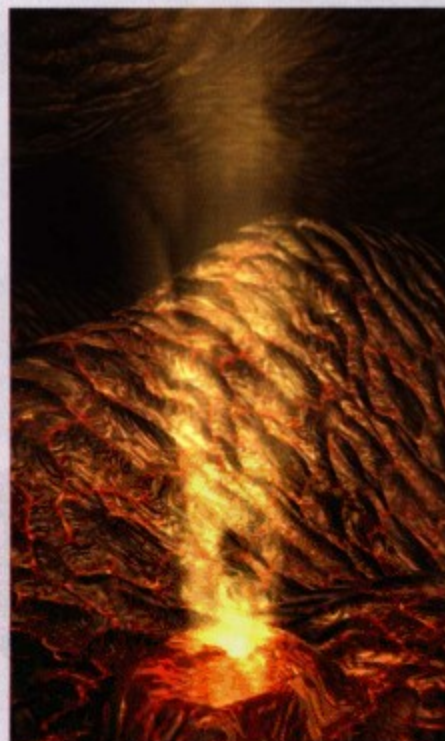


彩图 11

[左图]一个典型的使用漫射光照明的场景。虽然有正午阳光，但画面上的反差不够强烈。

[右图]同一个画面，但是根据取样得到的场景亮度对 gamma 进行了调整。

2003 Dave McCoy/Michael Dougherty 版权所有。使用已获得许可。



彩图 12

[左图]以线框方式绘制的热风口及热气模型。

[中间]按高度放大扭曲程度，消除生硬的边界。

[右图]最终得到的扭曲的画面。

2003 ATI Technologies, Inc. 版权所有。使用已获得许可。



彩图 18

利用圆形非均匀样条来生成游戏中的“旋转索道”可玩点。
2003 Krome Studios 版权所有。使用已获得 Krome Studios 许可。



内容提要

本书是著名技术丛书“游戏编程精粹”的第4卷，由全球数十位优秀游戏程序员撰写的文章汇集而成。书中有62篇长度中等难度适中的技术文章，分为通用编程、数学、物理、人工智能、图形图像、网络和多人游戏、音频共7章，并在随书光盘中提供了源程序和演示实现。文章的选题既紧跟游戏开发的时代脉搏，内容亦不流于表面，先进性和实用性俱佳。

本书适合游戏开发专业人员阅读，专家级开发人员可以立刻用书中介紹的方法和技巧，而初中级程序员通过阅读本书将增强其技能和知识。



序

作者：Mark DeLoura

E-mail: madsax@satori.org

感谢你选择《游戏编程精粹 4》！在这“游戏编程精粹”系列的最新一卷中，众多作者将一如既往地游戏程序员会遇到的大量问题进行探讨。或许他们中有人已经解决了你正在苦苦思索的技术难题。何必重复劳动呢？若你能（通过采用他们的解决方案）站在巨人的肩膀上，你就有机会成为下一位巨人。希望你对学会的算法进行扩展与改进，然后将你的结果公之于众。这样的知识传递，正是我们行业进步的关键所在，也是每一位游戏程序员对自己每天从事的工作保持不灭热情的关键所在。

为了反映你今日面临的挑战，我们在这一卷中增加了专门讨论游戏中物理问题的一章。随着游戏平台的机能日益强大，在游戏中实现实时物理的想法正广受关注。有一些正在开发的游戏已经开始无声无息地增加了一定的物理模拟，许多物体正开始按照玩家所能预料的那种符合物理原理的方式动作。从而，突显玩点（Emergent Gameplay）将成为可能，且为游戏增添前所未有的趣味。瞧，手里没有武器吗？那好，拾起一把折凳朝僵尸丢过去吧。

本书中的文字悉经 Andrew Kirmse 精心策划。早在 1996 年，Andrew 就参与开发了第一款 3D 多人在线角色扮演游戏 *Meridian 59*！现在 Andrew 是 LucasArts 公司的主程序员，开发的游戏如 *Star Wars: Starfighter* 等。他也是《游戏编程精粹 3》网络和多人游戏部分的编辑，并曾为“游戏编程精粹”系列的第一卷和第二卷撰稿。

业界的现状

当本书付印之时，我们正期盼着有关下一代游戏主机的信息。我们不由得猜疑，需要学哪些技术才能有效地在下一代主机上进行开发呢？然而，它们将带来的新的功能仍然是个谜。

近在眼前的是便携式游戏设备掀起的高潮。任天堂公司的 GameBoy® 已经以其各种型号独领风骚十多年了，但其他的公司正纷纷携各自有竞争力的产品进入这个市场。从 Nokia 的 N-Gage™ 到 Tapwave 的 Zodiac™ 到索尼的 PSP，也许微软心中也自有妙计，手持游戏市场已到了爆发的时候了。

时机选得正好，家用游戏机上的游戏软件开发费用昂贵而且风险也很

大，因此有另一种开发成本较低的平台无疑是件好事。如果你也是一名“蜗居”在车库中的游戏开发者，这对你一定是天大的好消息。今时今日，除了那些大发行商之外，又有谁能负担得起开发一个高质量游戏所需的至少 1500 万美元呢。

开发游戏的成本逐年上升，这在业界造成了不少麻烦问题，逐渐多见的炒冷饭问题只是其中之一。你又怎能埋怨游戏发行商不愿意承担风险呢？如果你有 1500 万美元钱用于投资，难道你不会倾向于较安全可靠的途径吗？今日的市场上，有着许多的续集、已有的游戏的克隆版以及各种各样的官方特许产品。这都是可以理解的，但并不令人满意。如果你是一名游戏玩家，重复玩雷同的游戏多半无法使你兴奋起来。

出路在何方？怎样才能把我们这些开发者从枷锁中释放出来，使我们有更大余地创新？我无法给你现成的答案，但我有个有益的建议。

贡献

常有人问我这样的问题，是什么原因让那些作者坚持为“游戏编程精粹”系列撰稿呢？难道他们不正是在将自己宝贵的技术成果泄漏给竞争对手么？这样问我，一定是从未在其他行业中见到类似的情况——可口可乐公司就绝对不会把它的绝密配方和百事可乐公司共享！那么，面对那些开发与你的游戏竞争的开发者 and 发行商，为什么你会将你的算法与之共享呢？

从这个角度看问题吧。如果你每次开发游戏都把代码从头写过，那就意味着每次都要重写保存游戏的函数库，对不对？可是你愿意每次都推倒重来吗？你一定会找个方法来节约时间，也免得让那些重复劳动逼得你发疯。那么在写完这个函数库之后，让你的朋友也使用它吧。也许你会问，我干吗要把自己辛苦好久才写出来的东西给别人用呢？答案是，你的朋友和你所处的情况一模一样。他已经重写了好多遍消除内存碎片的功能模块，这些重复劳动让他也一样很郁闷。如果你俩能共享各自的函数库，你们就都能事半功倍。这不是一件值得做的事情吗？

在较高层次来看，譬如说你为一家发行商工作，目前正在配合两家开发工作室工作。这两家工作室都正在从零开始编写全部代码。他们将各自编写功能完全相同的两套底层函数库。难道你不会倾向于将两者的开发工作在某种程度上予以合并，从而减少成本吗？一个游戏的基本模块并不会对游戏有太大的影响，何必要为两家工作室各自开发同样的东西而付双份钱呢？又譬如你为其中一家游戏工作室工作，合并部分模块工作的做法，将节约你的时间，使你能够更专注与朝游戏中增添 feature，而正是 feature 将使你的游戏鹤立鸡群。而且，既然发行商因其对工作室提供的服务而将抽取游戏销售收入中的大部分，那么若能在履行合同之余顺便获得一些有用的代码，不是很划算吗？

让我们在更高的层次看这个问题。我们是一个大行业，其中有着上千个游戏开发工作室。上千个工作室正在重复地开发着相同功能的函数库，这又是何苦呢？这正是诸如 STL 那样的库存在的理由：节约每个人的时间，也就等于节约每个工作室的开发成本。何不让这些工作室在某种抽象的层次进行协作，使其各自能将更多时间用来开发自己的游戏中独特的部分？今时今日，似乎无人有时间来创作革新的游戏，每个人都把如此多的时间耗在底层代码的重写上了（也变得与时俱进地复杂）以至于无法在日程中塞进更多创新的工作。

充分共享

于是你会说：好啊，那真是非常好。我只要等别人写完代码并“泄”（share）出来，随便拿来用就好。这样子一来，我既不用泄出一行自己的代码，又可以从中获益匪浅。对不起，这不符合游戏规则。若每个人都这么想，那就不会有会议、书、杂志等等的存在，也不存在任何性质的信息共享了。为了共享体系能够正常运行，每个人都应当出一份力。

你可以拿这个体系和税务体系相比拟。若没有人缴税，政府就会失去经济来源。道路龟裂不会有人来修，桥梁也会年久失修甚至断裂，冬天街上的积雪也不会有人从中开辟出通路。或许，你也能够自己来打理这一切，维护自己住的地方的附近区域。但是，更有效率的做法是缴一些税金，然后由各方面的专家为所有人解决一切。在软件开发上也是类似的。当你贡献出部分自己特别擅长的资源后，你就能从其他业界专家贡献出的资源中受益。

在一个税务制度完善的社会里，每一个要求权利但不愿同时履行缴税义务的人，在一定程度上侵犯了这个社会的利益。因为有这些拖欠税款的人的存在，其他人就需要缴更多的税。在技术领域也是类似的。这些不向群体做贡献的人，是拒绝分享自己的聪明才智。而同时，他人就不得不耗费较多时间，无谓地重写他们本来已经写好的完美的记忆卡函数库。在这种情况下，他们又有什么权利来抱怨业界缺乏创新精神呢？是你使得好多人不得不重头编写那可恨的记忆卡函数库！他们又怎会有时间来创新！

所以，简单说来这就是共享体系的根基，也是为何这对我们开发者团体来说是如此重要。在从他人的成果中获益的同时，每个向团体作出贡献的人也能得到各种形式的正面反馈，也许团体中的其他人进一步改进了相关代码，也许你获得了声望，甚至可能是一份酬金。

诚然，你也可能有一些不能与他人共享的商业机密，这是可以理解的。若你刚开发了一个效果美轮美奂的 pixel shader，那么等到你使用该 shader 的游戏发售之后再共享该 shader 也是可以理解的。事实上，在你的游戏发售以后，对于任何细心分析你的游戏的人来说，其中的技术就不再是秘密了。因此，何不公开共享呢？日积月累，这样做的人们积累了较多的信息，也提高了复杂度。这推动了许多像本书——《游戏编程精粹 4》一样的书的出现。精粹丛书之所以存在于今日，与全世界范围内巨大数量的支持者是分不开的。至今已有多达 200 人曾为游戏编程精粹系列撰稿，他们来自美国、加拿大、英国、法国、德国、瑞士、澳大利亚、巴西等地。显然，许多人都认同这一共享体系。

请认真考虑一下，你也能在游戏开发团体作出一份贡献。无论是向游戏编程精粹系列投稿、向杂志投稿、写书、在自己的网站上发表文章或披露部分你上一款游戏中的代码都行。比起什么都不做，只有给予能给你带来更多的获取。阅读他人的想法正是我们学习计算机科学的方式，也是未来的游戏开发者们学习实用有效的技术、从而避免重复开发的方式。这终将释放我们大家的创新力，带来前所未有的愉快体验。作为游戏开发者，也作为游戏玩家的一份子，这难道不正是我们真心希望的吗？



前言

作者：Andrew Kirmse, LucasArts Entertainment 公司

E-mail: ark@alum.mit.edu

在最近几年中我们见到了越来越多的有关游戏开发的实用书籍，其中不乏“游戏编程精粹”系列的功劳。本系列，包括你正在阅读的这一卷，是为数不多的专业游戏开发者向外人详细披露秘密的地方之一。我们希望你能体会文字背后的这些当今最先进、销量最大的游戏软件的开发经验，并将学来的技术和方法运用到你的日常工作中。

汇编一本实用的书，需要很好的平衡，既要有能够立即付诸实用的高端知识，也要能高瞻远瞩。由于游戏开发已变得非常专门化，我们特地邀请了 7 位业界专家来筛选和编辑 7 个主题的文章，他们是：

- 通用编程：Chris Corry, LucasArts Entertainment Company
- 数学：Jonathan Blow
- 物理：Graham Rhodes, Applied Research Associates, Inc.
- 人工智能：Paul Tozour, Retro Studios/Nintendo
- 图形图像：Alex Vlachos, ATI Research, Inc.
- 网络和多人游戏：Pete Isensee, Microsoft
- 音频：Eddie Edwards, Sony Computer Entertainment Europe

物理部分第一次在“游戏编程精粹”系列中出现，这也反映了在当前许多游戏中实时模拟和动力学的重要性的增长。曾几何时，物理模拟还只是飞行模拟游戏和驾驶模拟游戏的专利，但今日物理已在大多数出现交通工具及具有关节的角色的游戏中扮演着重要的角色。该部分的文章涵盖所有方面，从“游戏程序员的物理知识”直到有关物体实时变形的最新内容。

在本书中还有别的第一：第一位女性作者（恭喜你！）；更多的来自高校及研究院的稿件；在图形图像部分首次出现了有关最新硬件支持的 pixel 和 vertex shader 技术的讨论。将这一卷中的彩色插图与《游戏编程精粹 1》相比较，你就能看出在短短 3 年间，这一领域的进展是如何之快。阴影是特别热门的研究领域之一，在本书的图形图像部分详细介绍了新的阴影技术。有了新的硬件支持，一些极具视觉冲击力的新特效只需要要一点点小聪明就能做到。

如何利用本书



我们努力创造这样一本书，你会将它放在书桌上，而不是书架上。当你遇到新问题的時候，你会查阅本书，看是否这个问题已被他人解决。在多数情况下，你可以改编或者直接利用光盘上的源代码。如果你是某个领域的专家，你会愿意首先阅读对应的章节，但其他部分（特别是通用编程部分）可能也会包含对那个困扰你游戏已久的问题的解决方案。也可能你将在下一个项目中负责不同的部分，因此你会回来阅读本书中的其他章节。

在每一部分里，文章基本上是按照复杂度递增的顺序排列的。但编程老手们仍能从顺序靠前的文章中获益，新手也会找到顺序靠后的文章中所探讨方法的实际用途。当你逐章阅读下去的时候，这些特意安排的文章将使你越来越熟悉这个专题，其中涉及的数学知识也更为复杂。线性代数在数学和物理部分中特别有用，其他一些文章还用到了微积分初步、差分方程和数值计算方法。

随着本系列丛书涵盖的内容越来越广，用到的语言和第三方 API 的种类也越来越繁多。大多数代码是用 C++ 写的，此外也用到了一些解释语言，如 Java 和 Python。图形部分的文章涉及了 OpenGL、DirectX 以及各种 shader 语言。我们尽量以与特定 API 无关的方式描述问题与方法。

生生不息

有许多在几年前开始从事游戏开发的同行仍然记得那激发自己从事游戏开发的冲动的时刻。对我个人而言，一切都是从我六年级时懂得可以用一种叫“几何学”的神奇“玩意”在计算机屏幕上画圆开始的。从那时起，我们自学了许多书，那些书现在早已翻旧了，其中有高德纳先生(Donald E. Knuth)的 *The Art of Computer Programming*、Sedgwick 的 *Algorithms*、*Graphics Gems* 系列，也许还有 James D. Foley 和 Andries van Dam 合著的 *Computer Graphics* 一书中圆的 Bresenham 算法。就在数年之前，我曾在一次面试中被要求推导出该算法，这令我甜蜜地回想起六年级时在 Atari 800 计算机上绘制圆的情景。

如果你是一名专业游戏工作者，我希望此书能够唤醒你的求知欲。如果你刚刚起步，我希望你能够在本书中找到能够让你穷尽一生去追求和发现的美好事物。



致 谢

我衷心感谢《游戏编程精粹 3》的主编 Dante Treglia 和“游戏编程精粹”系列的主编 Mark DeLoura 给予我机会主编本书。Dante 在主编前一卷中积累下来的模版和建议，对我开展工作有莫大的帮助。Mark 创建了一个基于 Web 的应用程序，使得负责各章的编辑能用它来筛选文章，之后亦在编辑过程中不断地为我提供反馈。

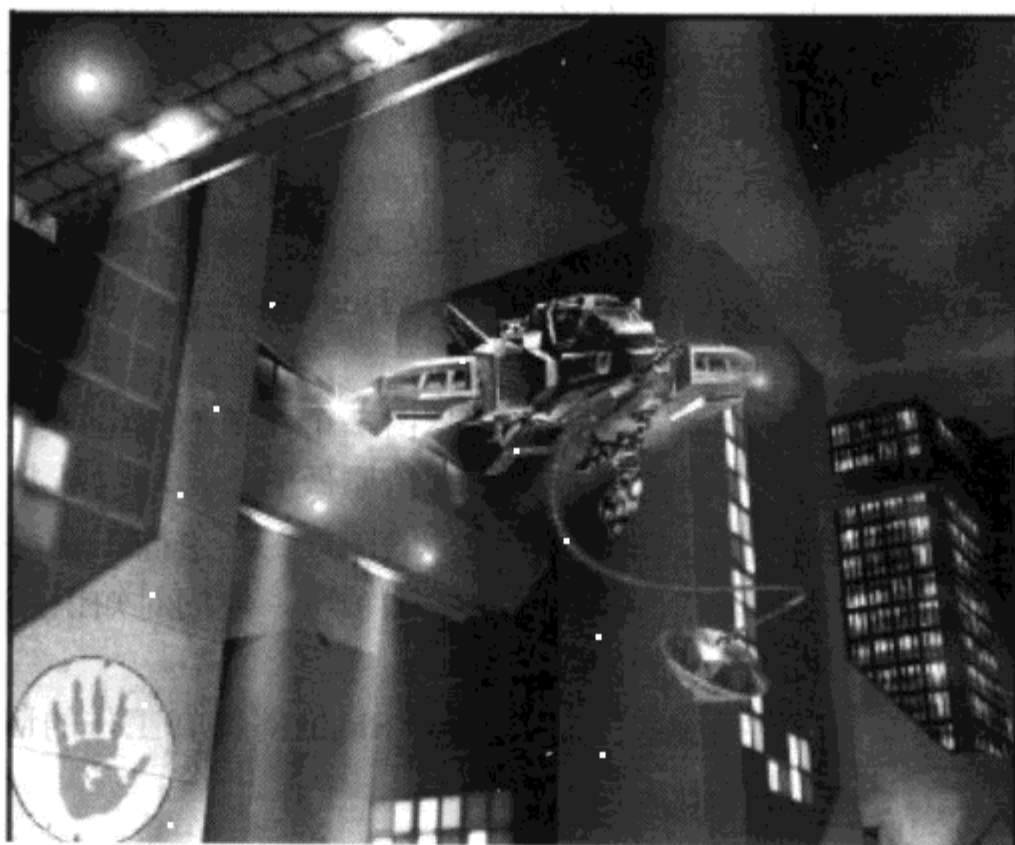
Charles River Media 出版社的同事们无一例外都是容易相处并且工作高效的。我主要的联系人是负责整个“游戏编程精粹”系列的出版人 Janier Niles。David Pallai、Meg Dunkerley 以及 Jennifer Blaney 也帮助回答了我的许多问题。

无疑，主要工作是由 7 个章节的编辑和数十位作者完成的。为本书这样的书撰稿是他们心甘情愿做的事，却也是非常费时的事。多位作者和编辑都热心地提供了额外的图片、彩色插图以及令人印象深刻的演示程序和示范源代码。在这个尖端领域，他们对知识共享的罕有的热情正是维系本系列丛书的纽带。

我还要感谢 Jennifer Sloan 对编辑工作的协助、在精神上的支持，以及她那无穷无尽的欢笑。

关于封面图案

作者：TJ Wagner, Day 1 Studios



封面展示的是游戏 *MechAssault II* 中的一个场景。该游戏是由 Day 1 Studios™和 Microsoft® Game Studios 为 Xbox®游戏机开发的。

游戏中所有的模型都是在 3ds max™中建模并用自定义的材质和贴图插件进行映射的。之后，游戏素材被导出并通过工作室自行开发的场景编辑工具放置在场景中。

MechAssault II 的图形渲染引擎完全是由 shader 驱动的，为了即时地汇编 shader 而使用了一个自定义的 shader 编译器和连接器。该引擎使用了多种渲染技术，包括逐顶点的光照、多通法向映射光照和亮度贴图。*MechAssault II* 还执行一些后期处理以便模拟动态的景深以及高度动态的光照。

游戏中出现的所有建筑物，都被设计成可由玩家动态破坏。破坏建筑物是通过即时地计算建筑物模型的形变来进行的。游戏中的建筑物可被严重破坏直至坍塌。建筑物的坍塌顺序是由设计师使用自制的特效工具来编写脚本控制的。

在封面所示的场景中，玩家控制的战斗机甲刚通过一条拖索挂上一架 VTOL，将要被快速运往敌方基地。老天保佑，这控制 VTOL 的 AI 或玩家可别把这部战斗机甲撞在建筑物上！

作者简介

Marwan Y. Ansari

mansari@ati.com

Marwan Y. Ansari 在 ATI 研究所的三维应用程序研究组工作。他曾在 DePaul University 获得计算机科学和数学的学士学位，后又在芝加哥 University of Illinois 获得计算机科学硕士学位。在加入三维应用程序研究组之前，他在 ATI 的数字电视组工作，更早的时候曾为 Number Nine Visual Technology 公司开发 OpenGL 驱动程序。除了本书收录的关于棕褐色色调转换的文章之外，Marwan 亦曾为 *ShaderX2* 一书撰文，并曾在 Game Developers Conference 上作有关使用 shaders 进行实时视频处理的演讲。

Jonathan Blow

jon@number-none.com

Jonathan Blow 居住在纽约市，是一位游戏技术顾问。他同时也为 *Game Developer Magazine* 的月度技术专栏 “The Inner Product” 撰稿。他还每年组织有关实验性 gameplay 的专题讨论会，展示各种新颖但具有市场风险的 gameplay，并激励更多新颖的 gameplay 设计出来。

James Boer

james.boer@gte.net

James Boer 的游戏生涯始于 *Deer Hunter* 游戏，（为了这款游戏）此后他一直默默忍受同事们善意的挖苦。James 工作过的其他游戏包括 *Rocky Mountain Trophy Hunter*、*Deer Hunter II*、*Microsoft Baseball 2000* 和 *Tex Atomic's Big Bot Battles*（网络游戏）。除了专业从事编程以外，James 也是一位多产的作者，包括撰写 *Game Developer Magazine* 中的一些文章，合著 *DirectX Complete* 一书，为“游戏编程精粹”系列的第一和第二卷撰稿，还有他的新书 *Game Audio Programming*。James 目前在 Amaze Entertainment 公司的 Black Ship 工作室工作，为目前主流和下一代游戏机开发跨平台的游戏。

Paul Bragiel

paul@paragon5.com

2000 年起 Paul Bragiel 任 Paragon Five 的 CEO。他公司的音乐工具曾在各种手持平台上被用来进行超过 25 个游戏中的音乐开发。此外他还担任过 3 个 GBC 游戏, 3 个 GBA 游戏和数个手机游戏项目的总制作人, 并曾在 GDC 和 Assembly conference 上作过数次报告。

Warrick Buchanan

warrick@chimeric.co.uk

Warrick Buchanan 在 Chimeric 公司担任开发主管, 产品包括 Maximina 和 ScreenSaverMax。他曾为不少游戏开发公司工作, 也曾为 Imagination Technologies 公司开发显示卡驱动程序。他所喜欢玩的东西从最先进的显示卡到蹦床, 无所不包。

Bill Budge

billbudge@hotmail.com

当 Bill Budge 还只有 3 岁大时, 他就知道建造一些东西会是自己今生的宿命。自打那时起, 他的玩具已从积木升级到了 C++ 的模板元编程, 但惊人地, 他仍然热衷于进行有趣的新形态构造。不编程时, 他喜欢和家人在一起, 喝从 Peet's 买来的混合咖啡饮料 Mocha Freddo, 或阅读计算机书籍。

Waldemar Celes

celes@inf.puc-rio.br

Waldemar Celes 是巴西里约联邦大学 (PUC-Rio, <http://www.puc-rio.br>) 计算机科学系的一名助理教授。他是一名研究人员, 同时也是该大学计算机图形技术研究小组 (Tecgraf/PUC-Rio) 的资深项目经理。此前曾在美国康乃尔大学 (Cornell University) 做计算机图形学方面的博士后工作。Waldemar 是 Lua 程序设计语言的发明人之一。他的主要研究兴趣是计算机图形学, 包括实时渲染、科学计算可视化、物理仿真、分布式图形应用程序等。

Chris Corry

chris@thecorrys.com

Chris Corry 是 Lucasfilm 公司旗下的 LucasArts 中的一名主工程师。他从事技术书籍的写作、合著、编辑足足已有十多年时间。《游戏编程精粹 4》是他的第 9 本合作结晶。

Carsten Dachsbaecher

dachsbaecher@cs.fau.de

Carsten Dachsbacher 是德国埃朗根—纽伦堡大学 (Universität Erlangen-Nürnberg) 计算机图形学专业的在读博士研究生。他的研究主要集中在交互式、基于硬件的计算机图形学领域，特别是阴影技术、基于点的渲染以及过程模型。他曾在德国 PC 杂志上连载 50 篇以上关于计算机图形学的文章。此外，他还承担一些为其他游戏公司开发 3D 游戏引擎的工作。

Mark DeLoura

madsax@satori.org

Mark DeLoura 是“游戏编程精粹”系列丛书的创办人。作为 Sony Computer Entertainment 美国公司的开发者关系经理，他有机会和遍布全世界的游戏开发者们共享技术性和非技术性的信息。Mark 非常执著于他的理念，也就是创造一个共享且愉快的体验，教育和鼓励游戏开发者相互交流。他曾担任 *Game Developer* 杂志的主编以及任天堂美国分公司的开发者支持组主软件工程师，期间他都一贯提倡“愉快的体验”。

Shekhar Dhupelia

sdhupelia@midwaygames.com

Shekhar 为 Midway Games 公司工作，主要负责网络游戏的策划与开发。之前 Shekhar 曾是 Sony Computer Entertainment America (SCEA) 的一名主程序员，作为 SCE-RT 组的核心成员之一，他曾花两年时间开发和设立了 Sony 的 PlayStation® 2 的在线平台及后端系统。他们的成果是诸如 *SOCOM: US Navy Seals*、*NFL Gameday 2003*、*Frequency* 等许多网络游戏不可或缺的部分。Shekhar 也曾在 High Voltage Software 公司工作，参与了与 Xbox™ Live 在线服务一同首发的微软 *NBA Inside Drive 2005* 的开发。他在 DePaul University 获得计算机科学学士学位。Shekhar 在 Midway 正在开发的游戏是 *NBA Ballers*。

Thomas Di Giacomo

thomas@miralab.unige.ch

Thomas Di Giacomo 的硕士学位论文是和 iMAGIS 实验室及 Atari (以前的 Infogrames) 的研发部共同完成的，讨论多分辨率的动画方法。他现在是瑞士日内瓦大学 (University of Geneva) 的图像处理实验室 (MIRALab) 的助理研究员和在读博士生。他主要从事动画的 LoD 以及基于物理的动画方面的研究。

Michael Dougherty

mdougher@hotmail.com

Michael Dougherty 目前在微软公司的 Xbox Advanced Technology 组工作。他专长于计算机图形学以及系统优化。Michael 在华盛顿大学 (University of Washington) 获得计算机工程学位。他希望在此对妻子 Jessica 和女儿 Elyse 对他工作所给予的无私支持和耐心表示衷心感谢。

George Drettakis

George.Drettakis@sophia.inria.fr

George Drettakis 于 1988 年在希腊的克里特岛大学 (University of Crete) 获得 Ptychion (相当于理学士) 学位。之后他在加拿大多伦多大学 (University of Toronto) 的计算机科学系师从 Eugene Fiume 完成了硕士学位 (1990) 和博士学位课程 (1994)。之后 (1994~1995) 作为欧洲信息及数学研究联盟 (ERCIM) 的博士后, 曾在法国格勒诺布尔的 iMAGIS、西班牙巴塞罗纳的 UPC 大学软件系 (LiSi) 以及德国 VMSSD-GMD 做访问研究。2000 年 7 月他迁至法国国家信息与自动化研究院 (INRIA) 的 Sophia-Antipolis 分院, 并在那里创建和领导了 REVES 研究组。目前的研究工作主要集中在阴影、光照、基于图像的模型重建以及交互式渲染等方面。

Eddie Edwards

eddie@tinyted.net

Eddie 初涉游戏开发是相继将 Wolfenstein 3D 和 DOOM 正式移植到阿基米德 (Archimedes) 计算机上。此后他曾为多家工作室工作, 有 Cranberry Source、Argonaut、Mucky Foot 和 Naughty Dog (SCEA 旗下)。在 Naughty Dog 时他是 *Jak & Daxter* 游戏中的底层代码工程师 (他开发了这款游戏的音频引擎)。Eddie 现在已回到英国, 主管 SCEA 旗下的 Psygnosis 公司的图形开发组。

David Etherton

etherton@rockstarsandiego.com

David 原先工作的 Angel Studios 在 1996 年被 Rockstar 买了下来, 因此就顺理成章地成了 Rockstar 圣地亚哥分公司的一员。他负责其所在工作室发布的大多数游戏中的渲染以及底层优化任务, 但他在 *Midtown Madness* 一代和二代、*Midnight Club* 一代和二代这几款游戏中出力最多。David 在公司里负责跨平台游戏函数库的技术事务, 其函数库是其公司 2000 年之后开发的游戏的基石。他也是研发部的主管。

Glenn Fiedler

gaffer@gaffer.org

Glenn Fiedler 知道有关高度场 (heightfield) 的一切知识, 目前正在 Irrational Games 公司倾其所知致力开发 *Tribes: Vengeance*。之前他曾完全重写了 *Freedom Force* 这款游戏用的寻路和物理子系统, 你猜改成什么了, 没错, 是一个 heightfield。Glenn 希望将来有一天, 能将自己已经很强的编程技能进一步拓展, 以便支持屋檐状的地形结构。

Peter Freese

pfreese@ncaustin.com

Peter 自 1990 年以来即活跃在游戏业界，那是从他决定自己与其为数据库程序员开发图形工具，不如去做游戏开始的。在开发了几款获奖的教育类游戏后，Peter 在 1994 年与其在 Edmark 的同事 Nick Newhard 一起创建了 Q Studios，之后主持了大受好评的 3D 射击游戏 *Blood*，使得发行公司 Monolith 一下子备受瞩目。在为 Sierra Studio 的最后一款冒险游戏 *Gabriel Knight 3* 开发了图形引擎之后，Peter 加入了 Origin Systems 公司的网络创世纪 2 开发组，在那里尝遍了开发多人网络游戏的酸甜苦辣。2001 年 Peter 为 NCsoft 在德克萨斯州奥斯丁创立了 Core Technology 组，在那儿他主持了最新的图形引擎的开发，NCsoft 的许多新网络游戏都使用他主持开发的引擎。

Bert Freudenberg

bert@isg.cs.uni-magdeburg.de

Bert Freudenberg 在德国马德堡大学 (University of Magdeburg) 攻读博士学位，研究方向是非照片写实风格的实时渲染技术。除了撰写论文及在 EuroGraphics 和 SIGGRAPH 等会议上作报告之外，他也参与了 OpenGL Shading 语言规格的制定和 *OpenGL Shading Language* (Addison-Wesley, 2003) 一书的写作。平时 Bert 喜欢小孩子，也喜欢教人有技巧的东西，不过他特别喜欢捣鼓一个叫做 Squeak 的跨平台环境。出于对 20 世纪 80 年代自己开始编程之后在计算机科学领域没有发生大的变化感到郁闷，他的最新兴趣转到了 OpenCroquet 操作系统环境上。

Paul Glinker

paul@glinker.com

受 Atari 2600 计算机的启蒙，Paul 在 7 岁时开始在一台只有 16KB 内存的 TRS-80 Co-Co-2 计算机上学习编程。在 13 岁时，出于对速度的追求，他开始在 8086 CPU 的计算机上用 C 语言编程。在高中时代，他最喜欢的计算机课教师鼓励他编写了自己的第一个类似街机的游戏，并放在当地的计算机商店中出售。他在加拿大 Laurentian University 获得计算机科学的 Honors 学位。他的教授总是喜欢讨论计算机科学的原理可以怎样被用于游戏开发，这对 Paul 的一生都有着很大的影响。在 2000 年他被 Rickstar Games 公司雇用为程序员，在那里工作至今。

Mario Grimani

mariogrimani@yahoo.com

Mario Grimani 早在两个世纪前便投身游戏业，是一名元老级人物。他曾在著名工作室工作，如 Ion Storm、Ensemble Studios、Verant Interactive 和 Sony Online Entertainment。在 Ensemble

Studio 时他是负责提高电脑控制的玩家厉害程度的 AI 专家。他开发了《帝国时代 2: *The Age of Kings*》和《帝国时代 2: *The Conquerors*》中的脚本系统以及计算机玩家的 AI。在神话时代开发的早期, Mario 担任首席 AI 程序员, 负责 AI 架构。在加入 Verant Interactive 后, 他担任 *Sovereign* 的主程序员, 尝试创造新的多人即时战略游戏类型。目前, 他在 Sony Online Entertainment 负责开发 *Everquest II* 中的一些模块。Mario 在克罗地亚的 University of Zagreb 获得电子工程专业的学士学位, 后在美国 Southwest Texas State University 获得计算机科学硕士学位。

John Hancock

jhancock93@post.harvard.edu

John Hancock 于 1999 年在 Carnegie Mellon University (CMU) 获得机器人学专业的博士学位。在 CMU 时, 他从事机器人汽车的视觉和导航算法研究。取得博士学位后, John 在 Activision 以 *Star Trek™: Armada* (于 2000 年 3 月发售) 的主程序员和 AI 程序员身份开始了其游戏生涯。在 2000 年 5 月他加入了 LucasArts, 开发 *Star Wars: Obi-Wan* 和 *Star Wars: Bounty Hunter*。他目前负责 *Star Wars: Republic Commando* 的小分队 AI。

Søren Hannibal

sorenhan@yahoo.com

Søren Hannibal 从 1985 年起就一直在编程, 当然除了必要的吃饭、睡觉和学习所占去的时间。Søren 最近完成了 *Enter the Matrix* 的动画系统, 开始感受到在 Shiny Entertainment 担任首席程序员的乐趣了——不时地鞭策一下懈怠的同事。

Matthew Harmon

matt@matthewharmon.com

Matthew Harmon 从念大学起就在开发游戏了, 在攻读电影理论和评论专业的学位的同时他也为 subLogic 公司的“微软飞行模拟器”游戏工作。从此以后, 他在 Mission Studios 公司和 Velocity Development 担任过首席程序员和开发主管。最近, 他与他人合伙 eV Interactive 公司, 继续开发游戏, 并在军事训练和模拟领域运用游戏开发的技术。在闲暇时, Matt 会绕着房子追自己两个淘气的儿子 Alex 和 Greg。

Oliver Heim

Oliver.Heim@intel.com

Oliver Heim 是 Intel 的图形核心开发组的资深软件工程师, 负责开发整合图形芯片的 Direct3D 设备驱动程序。之前 Oliver 曾在 Intel 的图形和三维技术组工作, 负责为 Shockwave3D 游戏引擎开发实时碰撞检测及物理算法, 以及研究包括光能辐射度、光线跟踪、量子映射在内的实时全局光照算法。在 1999 年之前, 也就是在加入 Intel 之前, Oliver 曾当过 3 年美国

Clemson University 虚拟现实实验室的主管。Oliver 在 University of Georgia 获得计算机科学学士学位，在 Clemson University 获得计算机科学硕士学位。在不忙的时候，你可以看见 Oliver 坐在门口弹吉他，或在加利福尼亚北部的某个瀑布顶上划着皮划艇冲下来。

Jim Hejl

jhejl@ea.com

Jim Hejl 是 Electronic Arts 的资深工程师，也是 EA 的 Tiburon Studio 的研发组成员之一。之前 Jim 曾参与开发 *John Madden Football* (2000-2003) 游戏。他非常喜欢太妃糖，但是比起工作来，他还是更爱工作。

Pete Isensee

pkisensee@msn.com

Pete Isensee 是微软公司 Xbox Advanced Technology 组的首席工程师。他擅长网络、性能和安全性方面的问题。Pete 拥有计算机工程学位。工作之余，他偶尔也会抛出几个“异常”，但什么事情也瞒不过他的眼睛。

Toby Jones

tjones@humanhead.com

Thobias Jones 是 Human Head Studios 开发 Xbox 和 PC 游戏的程序员。他在 University of Wisconsin, Platteville 获得计算机科学的学士学位。Thobias 从 PC 诞生起就一直在编程，出于兴趣自学了三维计算机图形学和计算机结构。要是不看动画也不编程的时候呢，Thobias 会通过他自己的公司 Genkisoft 出版一些共享软件。他的工作有所成就全靠亲爱的太太 Jess 支持。

Andrew Kirmse

ark@alum.mit.edu

Andrew 是 *Meridian 59* (1996) 的开发总监和设计者之一。他也是 *Star: Starfighter* (2001) 的图形程序员。他在麻省理工学院 (Massachusetts Institute of Technology, MIT) 获得物理、数学和计算机科学的学位。Andrew 曾为每一卷“游戏编程精粹”撰稿。他目前在 LucasArts 工作。

Adam Lake

adam.t.lake@intel.com

Adam Lake 是一名为坐落于美国俄勒冈州 Hillsboro 市的 Intel 微处理器研究实验室 (Microprocessor Research Labs, MRL) 工作的资深软件工程师，专长于下一代计算机图形架构和编程模型。他发表过一些文章，并在计算机图形学、虚拟现实、模型压缩、电子商务、

非照片写实风格的渲染等领域申请了不下 20 个专利。在进入 Intel 之前，他在美国查珀尔希尔的 University of North Carolina 研究计算机图形学和虚拟现实并获得计算机科学硕士学位。在那之前，他在美国 Los Alamos 国家实验室从事应用物理和理论物理，以及计算方法学的研究，参与一款叫做 Justine 的面向物理学家的计算机辅助设计软件的开发。详情请见 www.cs.unc.edu/~lake/vitae.html。在闲暇时间里，他是山地自行车手和公路自行车手、滑雪者、登山家、野营爱好者，也是书虫，但对开车则不在行。

Jay Lee

jlee@ncaustin.com

Jay Lee 在投身游戏业之前，已为 EDS 工作了 10 年之久，为 General Motors、Exxon 和 Sprint 这样的大客户提供 IT 服务。他首先加入了 Sierra 公司，参与了多款游戏的开发，如：*Gabriel Knight 2*、*Betrayal at Antara*、*Colliers Encyclopedia* 和 *SWAT 2*。之后他投身 Origin Systems 公司，初涉网络多人游戏世界。当时他是网络创世纪 2 (*Ultima Online 2*) 项目的数据库程序员和主要脚本编写者。目前 Jay 在德克萨斯州奥斯丁的 NCsoft Corporation 工作，担任 *Tabula Rasa* 游戏的主程序员和数据库程序员。

Noel Llopis

lllopis@convexhull.com

Noel Llopis 是 *C++ for Game Programmers* 一书的作者。他也多次为前几卷“游戏编程精粹”撰稿。他在 Day 1 Studios 刚刚完成了 *MechAssault* 的技术部分。目前正忙于研究及实现下一款游戏所需的技术。他关注游戏引擎的所有方面，从整体架构直到图形及碰撞检测。他在 University of Massachusetts Amherst 获得计算机工程的学士学位，并在位于查珀尔希尔的 University of North Carolina 获得计算机科学硕士学位。

Thomas Lowe

tomlowe@kromestudios.com

凭着对创造计算机游戏的一腔热情，以优异成绩毕业于英国 Warwick University 的 Tom 成为了澳大利亚布里斯班 (Brisbane) Krome Studios 的一名游戏程序员。他已在那里工作了 4 年，参与开发所有主要游戏机平台和 PC 上的各种类型游戏，包括冲浪类、动作类、platform 类。Tom 擅长动力学模拟、特殊效果、数学以及角色控制。

Frank Luchs

gameprogramminggems@visiomediamedia.com

早在 1983 年，Frank Luchs 就为 Atari 计算机编写了自己的第一个音乐程序，从而开始了他的音乐/编程的二重奏生涯。他工作过的项目包括电影和电视节目创作和编排乐谱，也包括

音响设计以及自定义工具和多媒体软件的编程。他曾创作和编排了数以百计的歌曲、韵律和电影配乐（包括德国最著名的系列剧《罪案现场（Tatort）》）。他创建了专门开发虚拟乐器产品的 Visiomedica 软件公司。在 Visiomedica，他设计了司芬克斯模块化媒体系统（Sphinx Modular Media System）。该系统是 Saccara™、Chephren™以及 Cheops™这些软件合成器的基础。Frank 在德国慕尼黑从事电影工作。不编程时，他非常喜欢摆弄自己的电子合成器，创作电子交响乐。

Nadia Magnenat-Thalmann

thalmann@miralab.unige.ch

Nadia Magnenat-Thalmann 教授在过去的 20 多年中一直领导虚拟人（Virtual Human）的研究。她拥有不同学科的数个学士和硕士学位，并于瑞士日内瓦大学（University of Geneva）获得量子物理学博士学位。1977 年到 1989 年中，她是加拿大蒙特利尔大学（University of Montreal）的一名教授。之后，1989 年她在日内瓦大学创立了图像处理实验室（MIRALab）。

Carl S. Marshall

Carl.S.Marshall@intel.com

Carl S. Marshall 是 Intel 公司未来平台实验室（Future Platform Labs）的资深软件工程师。他在 Clemson University 主持虚拟现实的研究，并在那里获得计算机科学硕士学位。他曾在“游戏编程精粹”第二卷和第三卷上发表文章。之前他参与开发 Shockwave3D 图形引擎中的非照片写实渲染（NPR）及其他一些模块。目前，Carl 感兴趣的有写实主义的实时图形引擎、未来的硬件架构等。

Adam Martin

gpg@grexengine.com

10 多年来，Adam 深深地着迷于虚拟世界及实现虚拟实境所需的相关技术——图形、计算、分布式系统等。他在英国剑桥大学获得计算机科学学位，曾两度在 IBM 的研究实验室中工作，并共同创建了一家 IT 咨询公司。在剑桥商业计划竞赛中，他和他的队友曾获得 5 万欧元之多的奖金，从此他一直热心为刚起步的新公司介绍经验、提出建议。在 2001 年，他创办了 Grex Games 公司，将其之前的研究具体化为 MMOG 的开发和中间件技术。他还曾就 GrexEngine 的架构做过数次会议报告。

Maic Masuch

masuch@isg.cs.uni-magdeburg.de

Maic Masuch 在德国马德堡大学（University of Magdeburg）工作，是德国的首位计算机

游戏专业的教授。他从 1996 年起开始执讲计算机游戏开发课程。从那以后，许多学生的游戏项目在他的指导下完成，其中有简单的冒险游戏，也有非暴力的多人 FPS 游戏，更有一些交互式虚拟实境（VR）设备研制。他的研究主要集中于游戏开发的方法和工具，以及新的用户界面。目前他正在主持以下研究：智能开发工具、电影的编剧手法、非写实风格的渲染、自动 gameplay 分析等。Maic 也是 Impara 公司的创始人之一。Impara 主要开发面向儿童教育的教育类媒体创作工具和娱乐系统。

Dave McCoy

david.mccoy@comcast.net

Dave McCoy 是一名画匠，目前在微软公司的 Xbox Advanced Technology 组中工作。自 1991 年进入公司以来，他曾先后担任过艺术总监（Art Director）、设计师、制片人（Producer）等职。他的作品曾被多本专门介绍游戏图形的书籍和期刊所引用。他还拥有两份关于绘画方法的专利。

Ádám Moravánszky

adam.moravanszky@novodex.com

Ádám Moravánszky 是他本人一手创建的物理中间件公司 Novodex AG 的核心技术部门的首席工程师。他曾在匈牙利、德国、美国等地接受教育，最后在瑞士联邦技术学院（Swiss Federal Institute of Technology，简称 ETH）的苏黎世分院获得计算机科学的硕士学位。

Frederic My

fmy@fairyengine.com

1985 年，还只有 14 岁的 Frederic 开始学习编程，并出于兴趣在一台 TO7-70 和一台 Amstrad CPC 电脑上编写了几个游戏。20 世纪 90 年代中期，他几乎把全部时间都用来和天才好朋友 Alexis Vaginay 一起在 PC 上编写 Demo。从学校毕业之后，他投身游戏业，用汇编和 C++ 语言编写 3D 引擎和工具。若你见到 Frederic 的电脑关着，他不是跑步、骑车就是在重温电视剧《僵尸猎人：芭斐》。

James F. O'Brien

job@eecs.Berkeley.edu

James F. O'Brien 是加利福尼亚大学伯克利分校（U.C. Berkeley）计算机科学系的教授。他著有 30 篇以上的期刊和会议论文，其中 8 篇在声望很高的 ACM SIGGRAPH 会议发布。除了在 U.C. Berkeley 的执教经历外，他还进行一些技术讲座，并在游戏开发者年会（Game Developers Conference, GDC）和 SIGGRAPH 教授一些课程。他的主要研究方向是基于物理的模拟在离线动画和交互动画。

Chris Oat

Coat@ati.com

Christopher Oat 是 ATI 研究所的三维应用程序研究组的一名软件工程师，负责为实时三维图形应用程序开发新颖的渲染技术。他的工作中心是为现有的和将来的图形平台开发 Pixel 和 Vertex Shader。Chris 原是 RenderMonkey 开发组的一员，最近也为 ATI 的演示程序和屏幕保护程序编写 shader。他写的一些关于高级渲染技术的文章，在《游戏编程精粹 3》、《游戏编程精粹 4》、*ShaderX* 和 *ShaderX2* 等书中都有收录。Chris 毕业于美国波士顿大学 (Boston University)。

John M. Olsen

infix@xmission.com

1989 年 John M. Olson 毕业于 University of Utah，但他在之前就开始开发各种图形程序和软件了。John 目前在微软开发 Xbox 游戏。他多次撰写文章向讨论计算机图形学和游戏开发的书投稿并获得采用，包括“游戏编程精粹”系列和 *Massively Multiplayer Game Development*。他也曾在游戏开发者年会 (GDC) 发言。兴趣包括自治 AI、全息图像生成、网络和数据的组织与分析。

Marcin Pancewicz

highway@idreams.com.pl

Marcin Pancewicz 是波兰 Infinite Dream 公司的全职程序员。他负责开发二维游戏引擎以及游戏开发所需的工具。他住在波兰西南部的格莱维茨市。在不写程序的闲暇时间里，他喜欢养护他的爱车——一台已有 14 年历史的 Trans Am 跑车。

Kurt Pelzer

kurt.pelzer@gmx.net

Kurt Pelzer 是 Piranha Bytes 公司的软件工程师，他参与开发的 PC 游戏有 *Gothic*、销量冠军 *Gothic II*（这两款游戏各自被评为德国 2001 年度和 2002 年度的“最佳 RPG 游戏”），以及资料片 *Gothic II: The Night of the Raven*。之前他是 Codecult 公司的高级程序员，基于公司的高端 3D 引擎“Codecreatures”开发了数个实时模拟和技术演示程序（例如：为西门子公司开发的上海磁悬浮机场快线的模拟，以及著名的显示卡性能基准测试软件 *Codecreatures Benchmark Pro*）。他也曾发表文章于 *ShaderX2* 和 *GPU Gems*。

Borut Pfeifer

borut_p@yahoo.com

1998 年 Borut Pfeifer 于美国 Georgia Institute of Technology 毕业。在担任过各种软件开发职位后, 2001 年 5 月时他与朋友一起创立了 Knuckle Games 公司, 自己担任首席设计师和 gameplay/AI 程序员直到 2003 年 4 月。目前他在加拿大温哥华的 Radical Entertainment 公司工作, 也著有一些游戏开发文章。

Karén Pivazyan

pivazyan@stanford.edu

Karén Pivazyan 是一名有 9 年咨询经验的游戏 AI 架构师。他擅长于运用最新的学术研究成果解决游戏 AI 中的难题。他 6 岁开始玩游戏, 10 岁就开始了制作游戏, 拥有麻省理工学院和斯坦福大学的学位。他的博士研究是有关 multi-agent 系统学习的。他的理想之一是用心理学建立人类行为的模型。他打算开始授课并写一本有关游戏 AI 原理的书。

Nick Porcino

nporcino@lucasarts.com

Nick Porcino 是 LucasArts 的 R2 图形组的负责人。Nick 制作的第一款计算机游戏发布于 1981 年, 那是一个在 Apple II 型计算机上运行的游戏。Nick 的第一款游戏机游戏则是在 1984 年开发的 ColecoVision™ 上的卡带游戏。他曾数次试图离开游戏业, 曾在加拿大的 Royal Roads Military College 为无人潜艇设计过 AI, 也曾开发建筑可视化 VR 模拟, 还曾为日本 Bandai 公司设计过玩具、巨型机器人、直线加速器。不过, 一次又一次地, 他总是又回到游戏业。但这次他不会离开了, 能为下一代平台开发共享的核心技术让 Nick 非常心满意足。

Mark T. Price

mark@suddenpresence.com

Mark 开发游戏的年头几乎与个人电脑的历史等长, 1979 年在 CP/M 系统上起步, 经历了苹果电脑和 Atari 的岁月, 接着是 PC, 目前是 GameBoy Advance。可惜, 他并非全职开发游戏软件。1987 年, 当 Mark 大学毕业时, 以游戏开发为职业的前途似乎并不光明, 因此他选择了一份“普通的”工作。他在路透公司 (Reuters Ltd.) 工作过许多年, 开发了实时网络证券市场信息系统, 遍布美国和加拿大的超过 6 万个证券经纪人都使用这个系统。2001 年 Mark 感到是时候重回游戏业的怀抱了, 便与人联手创立了 Sudden Presence 这家承接游戏和多媒体产品订单的公司。他目前担任此公司的首席科学家。

Matt Pritchard

mpritchard@ensemblestudios.com

Matt Pritchard 是 Ensemble Studios 的高级工程师, 参与了 Age of Empires 系列游戏的开发。这些日子他在开发一款未公开的游戏。他反对在线游戏中的作弊行为。平时则喜欢陪儿

女、做家务，还喜欢摆弄汽车。他相信一台客货两用车（Station Wagon）飚到 175 英里每小时并不是绝对不可能的事。

Justin Quimby

justin@turbinegames.com

Justin Quimby 是涡轮娱乐软件公司（Turbine Entertainment Software）的首席工程师。在过去 5 年中，这名布朗大学（Brown University）的毕业生已在不少游戏项目中工作，包括 Asheron's Call 系列中的每一款，还有《指环王：中土在线》（*The Lord of the Rings: Middle-Earth Online*）。目前的 Quimby 正在圆着小时候开发在线龙与地下城游戏的梦想。在为 Turbine 公司打拼多人网络游戏市场份额之余，Quimby 热衷于阅读有关齐伯林飞艇（Zeppelins）的一切，以皇帝的名义征伐兽人部落，也喜欢与陌生人聊天。

Steve Rabin

steve@aiwisdom.com

Steve Rabin 已置身游戏业十余载，目前在任天堂美国分公司工作。他曾为 3 款已经发行的游戏开发 AI，也曾为“游戏编程精粹”系列目前为止的全部 4 卷撰稿。他是《游戏编程精粹 2》人工智能部分的编辑，之后更创办了“人工智能游戏编程真言”系列丛书，担任第一和第二卷的总编，亦曾在 GDC 上多次就人工智能进行演讲。他在华盛顿大学（University of Washington）获得计算机工程的学士学位，专业是机器人学。目前，工作之余的 Steve 还同时在华盛顿大学攻读计算机科学的硕士学位。

Graham Rhodes

grhodes@nc.rr.com

Graham 从事游戏编程中各方面的工作已有 20 年的经验，包括计算几何学、实时三维图形学以及物理。他现在是 Applied Research Associates 公司的首席科学家。在 ARA 工作期间，Graham 开发了曾被教育类游戏 *WorldBook Multimedia Encyclopedia*（Windows 光盘版）中几个小游戏（mini-game）采用的游戏引擎。其中一款小游戏以实时模拟玩具滑翔机和飞盘（Flying Disc）为例来讲解控制飞行的原理。Graham 还曾主持 NASA（美国航空航天局）的 NextGRADE SAM 计划（全称是 Next Generation Revolutionary Analysis and Design Environment Smart Assembly Modeler）。Graham 为《游戏编程精粹 2》写过一篇文章，也曾在 GDC 和 XGDC 游戏开发者会议上发言。目前 Graham 还在开发一款为海上石油钻井平台的乘员而开发的第一人称视角的安全训练游戏。

Thomas Rolfes

tr@circensis.com

Thomas Rolfes 曾在 Criterion 和 Kuju 公司的游戏和引擎开发组中工作，也曾在 Lionhead

Studios 担任引擎主程序员,参与开发了多款引擎和游戏,如 *Redline Racer Dreamcast*、*Microsoft Train Simulator*、*Lotus Challenge PS2/Xbox* 以及 *Black & White 2*。他拥有德国明斯特大学 (University of Münster)、哈根大学 (University of Hagen) 的物理和计算机科学学位。Thomas 曾参与创作一本有关行星轨道的书,不忙着写 shader 代码的时候他会自己动手造望远镜。

Greg Seegert

gseegert@alum.wpi.edu

Greg Seegert 是 Stainless Steel Studios 的图形程序员。他曾参与开发 *Empires: Dawn of the Modern World* 和 *Empire Earth* 两款游戏。Greg 写的程序个性张扬。他总在试图说服工作室里的头头们让他负责开发最新最好的 Nvidia 或 ATI 的技术演示程序 (Tech Demo)。他打算办一个接受旧电脑捐赠的慈善机构,这样他就不用担心有朝一日发现自己身边已经到处都是 5 年前的主流 CPU 和 GPU 了 (大多数 RTS 游戏爱好者都有这么一天)。

Jake Simpson

jmsimpson@maxis.com

Jake Simpson 喜欢说自己已经从事游戏开发很久了。但是他没可能在 20 世纪 60 年代参加 Eliza 项目的开发,因为他妈妈直到 1968 年才生下他。不过,他确实曾在 Midway 芝加哥分公司工作了 6 年,开发街机游戏 (为了使自己的名字出现在 *Revolution X*、*WWF Wrestlemania*、*NBAJAM Nani Edition*、真人快打 *Mortal Kombat III* 等游戏的 Credit 画面上)。他也在 Raven Software 工作过 4 年,期间开发了 *Heretic II*、*Soldier of Fortune I & II*、*Star Trek Voyager: Elite Force* 和 *Jedi Knight II* 等游戏。他目前在 Maxis 工作。尽管工作十分卖命,但是要在 *The Sims 2.0* 中做一杆双管暴力枪 (Shotgun) 的“小阴谋”迄今为止还没有获得成功。

Roger Smith

roger@fingersofdeath.com

Roger Smith 是 Titan 公司的首席技术官 (CTO),也是 Modelbenders 有限公司的总裁。他为军队开发模拟系统,提供模拟及虚拟环境技术的课程,常常撰写技术文章向会议论文集和书投稿。他在 GDC 上做讲座已经连续 4 年了,还定期在数所大学授课。他和 Don Stoner 一起维护着的死神之指 (Finger of Death) 网站主要是为了表达自己对这一领域的爱好。

Russ Smith

russ@q12.org

Russ Smith 是 Open Dynamics Engine (ODE) 的作者。ODE 是一个模拟以关节连接的刚体动力学的开源函数库 (Open-source Library)。Russ 还曾参与开发 MathEngine 的游戏开发相关产品,主持两足直立行走机器人的模拟,还为电影制作模拟的虚拟角色。

Marco Spoerl

mspoerl@gmx.de

Marco 开始其计算机图形的职业生涯是在 Codecult Software 公司担任引擎程序员。他参与了 Codecreatures 游戏开发平台系统和显示卡性能基准测试软件 Codecreatures Benchmark Pro 的开发。在获得计算机科学学位后, Marco 曾自由自在地当过一阵子自雇程序员, 现在的他在德国慕尼黑的克劳斯—玛菲—威格曼 (Krauss-Maffei-Wegmann, 简称 KMW) 军火公司的训练与模拟部工作。

Marc Stamminger

stamminger@cs.fau.de

Marc Stamminger 在德国埃朗根—纽伦堡大学 (University of Erlangen-Nurnberg) 获得计算机图形学专业的博士学位。他的论文题目是全局光照计算中的有限元方法。此后, 他曾在位于德国萨尔布吕肯市 (Saarbrücken) 的马克斯-普朗克计算机科学研究所 (Max-Planck-Institute for Computer Science)、以及法国国家信息与自动化研究院 (INRIA) 的 Sophia-Antipolis 分院进行博士后研究。当开始接触阴影算法和基于点的渲染之后, 他的研究方向开始转向交互计算机图形学。2002 年时他在德国魏玛鲍豪斯大学 (Bauhaus University) 担任计算机游戏讲师。2002 年 10 月之后他成为了埃朗根—纽伦堡大学的计算机图形学和可视化的教授。

Jonathan Stone

jon@doublefine.com

Jonathan Stone 用 Atari 800 计算机上的 BASIC 语言编写了他的第一个视频游戏, 1992 年之后他成为了一名职业游戏程序员。他最近完成并发售的游戏是暴雪北方工作室 (Blizzard North) 的《暗黑破坏神二代》(*Diablo II*)。目前他在 Double Fine Productions 公司开发 Xbox 游戏机上的动作游戏《疯狂意识世界》(*Psychonauts*)。

Don Stoner

don@fingersofdeath.com

Don Stoner 是 Titan 公司的软件工程师。他为军队开发模拟系统, 也开发过事件模拟可视化和网络模拟的工具, 目前正为无线电系统的训练机开发语音识别系统。他是前军情局和特种部队成员, 有实战、信号截取、侦查、监视等方面的经验。是 Roger Smith 选中了他一起写作《死神之指》一文, 但他逐渐也真的对这个主题入迷起来。

Thomas Strothotte

tstr@isg.cs.uni-magdeburg.de

Thomas Strothotte 是德国马德堡大学 (University of Magdeburg) 的一名教授, 1993 年起他是图形学和交互式系统专业的系主任。他的研究方向包括智能图形、图像和文字的一致性、示意图渲染、非照片写实风格的渲染、虚拟社区, 应用在医学图解、技术文档和计算机游戏中。Thomas 在加拿大蒙特利尔的 McGill 大学获得计算机科学博士学位。

Natalya Tatarchuk

natashat35@yahoo.com

Natalya Tatarchuk 是 ATI 研究所的三维应用程序研究组的一名资深软件工程师。她担任 RenderMonkey 集成开发环境项目组的组长, 负责开发使实时的 shader 程序变得容易开发的工具软件。她已在图形领域工作了 6 年多, 在加入 ATI 之前她主要从事三维建模程序和科学可视化的工作。Natalya 曾在 *ShaderX2* 一书中发表文章, 也曾出席 Microsoft Meltdown Seattle、GDC、GDC Europe 等会议。她毕业于波士顿大学 (Boston University)。

Pierre Terdiman

pierre.terdiman@novodex.com

Pierre Terdiman 已从事程序设计工作 15 年多了, 他的编程生涯从在 Atari ST 计算机上编写 demo 开始, 全屏幕的编程让他感受到乐趣 (现在也让他感伤)。在 RAYflect (今天的 EOVI), Pierre 负责扫描线渲染和 A-buffer 抗锯齿的实现。他的一些个人项目今日已在不少游戏和图形软件公司中得到使用, 其中包括 Flexporter 和 Opcode。之后, Pierre 为 Elsewhere Entertainment 公司的一款即将推出的游戏 *Symbiosis* 开发了物理引擎。2001 年 Pierre 创办了他自己的公司 Synthetic3, 专门开发 Web 上的三维技术。此后他加入了 NovodeX, 目前负责开发高级碰撞检测和物理函数库。

Jerry Tessendorf

jerryt@rhythm.com

Jerry Tessendorf 是 Rhythm & Hues 工作室的一名特效技术指导。他开发和运用水模拟软件已长达 20 年, 负责多部电影大片中的水的特效软件, 这些电影包括《泰坦尼克号》(*Titanic*)、《未来水世界》(*Waterworld*)、*X2: X-Men United* 等。他的实时水模拟除了被用于游戏以外, 还被用于国防模拟程序。除了和水有关的以外, 他还开发了许多渲染云彩、粒子和毛发的自定义算法及软件。Jerry 喜欢钻研光线追踪算法、图形学中的微分几何学 (Differential Geometry) 以及环圈量子重力论 (Loop Quantum Gravity)。他在布朗大学 (Brown University) 获得物理博士学位。

Paul Tozour

ptozour@austin.rr.com

Paul Tozour 自 1994 起就一直一直在游戏行业工作, 之前他曾在 Red Orb Entertainment、Gas

Powered Games、Microsoft Game Studios、Ion Storm 等工作过。目前他在德克萨斯州奥斯丁的 Retro Studio 工作，是 *Metroid Prime 2: Echoes* 项目中的一个 AI 开发者。他也曾为“游戏编程精粹”系列和 *AI Game Programming Wisdom* 系列丛书撰写了不少关于人工智能的文章。

Joe Valenzuela

jvalenzu@infinite-monkeys.org

Joe Valenzuela 是 Treyarch (Activision 的子公司) 的程序员。他也是 OpenAL 的 Linux 实现版本的主要作者和维护者。

Jim Van Verth

jimvv@redstorm.com

Jim Van Verth 是 Red Storm Entertainment 的创始人之一，他已经为之工作了 7 年。其中的大多数时间，他作为主工程师，主持像 *Tom Clancy's Politika* 和 *Force 21* 这样的游戏开发项目。他在 UNC Chapel Hill 获得计算机科学的硕士学位，在那里他学习了科学可视化和计算机图形学。他家房子很偏僻，饱受风吹雨淋，周围岩石嶙峋，他和妻女还有 500 条饿狗一起住在那里。但有时他的夸张也是出了名的，其实他家只养了一条狗。

Scott Velasquez

scottv@gearboxsoftware.com

Scott Velasquez 是 Gearbox Software 公司的音频和游戏程序员。他曾开发过 *Counterstrike: Condition Zero*、*Nightfire* 等游戏，现在正在开发 *Halo* 的 PC 版本。Scott 拥有软件工程的学士学位。在加入 Gearbox 之前，他在 Cinematix Studio 工作，研发 PS2/PC 引擎。在 Cinematix 时他负责开发音频引擎、多视角摄像机，甚至涉及其渲染系统。工作之余，Scott 通常和他的妻女在一起。

Alex Vlachos

Alex@Vlachos.com

Alex Vlachos 于 1998 年加入 ATI，目前是三维应用程序研究组的工程师。作为 ATI Demo 组的主程序员，他主要负责 3D 引擎的研发。N-Patches（一种曲面表示法，在微软 DirectX 8 中得到使用），也称为 PN Triangle 或 TRUFORM，便是他参与开发的。他在“游戏编程精粹”的 1、2、3、4 卷，交互式 3D 图形（Interactive 3D Graphics，简称 I3DG）的 ACM 研讨会以及 *ShaderX* 和 *ShaderX2* 上都发表过文章。他还多次在 GDC（游戏开发者会议）、GDC Europe、Microsoft Meltdown Seattle & UK、WWDC 和 I3DG 等会议上做过报告。Alex 毕业于波士顿大学。可以通过 <http://alex.vlachos.com> 与他取得联系。

Tao Zhang

zhangtao@cc.gatech.edu

Tao Zhang 在北京大学获得计算机科学学士学位。目前正在美国 Georgia Institute of Technology 攻读博士学位并从事编译器、体系结构和安全等方面的研究。自从迷上计算机游戏后，他一直希望能对游戏开发社区有所贡献。



翻译和审校人员

- 沙鹰** shaying@babafish.com
负责《游戏编程精粹 4》全书统稿、定稿。
翻译了 1.1~1.7, 1.9~1.12, 4.1、4.6 和 4.8 共 14 篇文章。
审校了 3.1~3.3、6.1、7.1、7.4 以及整个数学部分共 12 篇文章。
目前在 EA Canada 工作。之前曾在 JAMDAT Canada 和上海育碧工作。
参与开发的游戏主要有 FIFA Street、Splinter Cell、Ghost Recon、Sum of All Fears 和 VIP 等。沙鹰也是《Windows 游戏编程大师技巧（第二版）》的译者，毕业于南京大学计算机系。
- 刘永静** yongjingleu@hotmail.com
翻译了整个图形图像部分共 15 篇文章。
审校了 1.7、1.9、3.6~3.8 共 5 篇文章。
Jacky 在上海盛大历任技术保障中心游戏运营项目经理、研发中心项目经理。
Jacky 也是中国游戏开发中心网站（OGDEV）的创始人之一。
- 许竹钧** antidreamer@hotmail.com
翻译了 6.1~6.5 以及整个数学部分共 11 篇文章。
审校了 1.11、1.12、3.4、3.5、7.2 和 7.6 共 6 篇文章。
James 先后在南京大学和加拿大 University of Guelph 获得计算机科学的学士和硕士学位。
- 万太平** taipingwan@msn.com
翻译了 1.8、1.13 以及整个音频部分共 8 篇文章。
审校了 1.1~1.6、1.10、6.2~6.4 共 10 篇文章。
Andy 在杭州电子科技大学（原杭州电子学院）获得机械电子工程学士和计算机科学硕士学位，目前在上海光通公司工作。在加入光通之前，Andy 在广州网易从事 3D 游戏的开发工作。感兴趣方向：小脑模型神经网络（CMAC）、图形及数字图像处理等。
- 李鸣渤** limingbo@msn.com
翻译了整个物理部分共 8 篇文章。
审校了 7.3 一文。
Frank 在英国 Liverpool John Moores University 获得计算机游戏技术专业硕士学位。其后在英国纽卡斯尔的 Eutechnyx 公司担任工具及游戏程序员，参与游戏 Street Racing Syndicate 的开发。专业兴趣包括游戏中的物理模拟和 Vertex/Pixel

Shader。

肖罡

camelxg@hotmail.com

翻译了 4.2、4.3、4.4、4.5 和 4.7 共 5 篇文章。

审校了 6.5 一文。

肖罡先后在南京大学和加拿大 University of Alberta 获得计算机科学的学士和硕士学位，目前在 EA Canada 工作。

石卫东

shiw@cc.gatech.edu

负责 6.6 一文。

Larry Shi 在美国 Georgia Institute of Technology 攻读博士学位。作为 6.6 一文的原作者之一，Larry 将其译为中文。

感谢袁正敏 (yuan_z_m@hotmail.com) 对物理部分翻译工作的协助。

特别感谢下面三位 EA Canada 的软件工程师对本书校对工作的大力协助。

李劲松

ljs_aaa@msn.com

校读了整个人工智能部分共 8 篇文章。

目前在 EA Canada 工作。之前曾在上海育碧工作。

参与开发的游戏主要有 FIFA 和 Rayman 等。格言是“少写程序多散步”。

谷超

guchao@msn.com

校读了整个图形图像部分共 15 篇文章。

目前在 EA Canada 工作。之前曾在上海育碧工作。

参与开发的游戏主要有 FIFA、NBA Street 和 Rainbow Six 等。

肖丹

emma.xiao@mail.com

校读了 1.8、1.13 和 7.5 共 3 篇文章。



目 录

第 1 章 通用编程

简介	2
<i>Chris Corry</i>	
1.1 调试游戏程序的学问	4
<i>Steve Rabin</i>	
1.1.1 五步调试法	4
1.1.2 第一步：始终如一地重现问题	4
1.1.3 第二步：搜集线索	5
1.1.4 第三步：查明错误的源头	6
1.1.5 第四步：纠正问题	7
1.1.6 第五步：对所作的修改进行测试	7
1.1.7 高级调试技巧	8
1.1.8 困难的调试情景和模式	10
1.1.9 理解底层系统	12
1.1.10 增加有助于调试的基础设施	12
1.1.11 预防 bug	13
1.1.12 结论	14
1.1.13 致谢	15
1.1.14 参考文献	15
1.2 一个基于 HTML 的日志和调试系统	16
<i>James Boer</i>	
1.2.1 于日志系统的优势	16
1.2.2 究竟什么是事件日志？	16
1.2.3 HTML 和调用堆栈	17
1.2.4 工作原理	18
1.2.5 一些有用的心得	21
1.2.6 结论	21
1.3 时钟：游戏的脉搏尽在掌握	23
<i>Noel Llopis</i>	
1.3.1 关于时间的基础	23
1.3.2 时钟系统的组成	24
1.3.3 避免失真	25
1.3.4 结论	29
1.4 设计和维护大型跨平台库	30
<i>David Etherton</i>	

1.4.1	设计	30
1.4.2	Build 系统	32
1.4.3	细节	33
1.4.4	结论	35
1.4.5	参考文献	35
1.5	利用模板化的空闲块列表克服内存碎片问题	36
	<i>Paul Glinker</i>	
1.5.1	内存操作	36
1.5.2	解决方案	37
1.5.3	实现细节	37
1.5.4	有效地使用我们的 Freelist	40
1.5.5	结论	40
1.5.6	参考文献	41
1.6	一个用 C++ 实现的泛型树容器类	42
	<i>Bill Budge</i>	
1.6.1	可重用的库	42
1.6.2	树的概念	43
1.6.3	树的实现	43
1.6.4	利用 STL	46
1.6.5	结论	49
1.6.6	参考文献	49
1.7	弱引用和空对象	51
	<i>Noel Llopis</i>	
1.7.1	使用指针	51
1.7.2	弱引用	52
1.7.3	空对象	55
1.7.4	结论	56
1.7.5	参考文献	57
1.8	游戏中的实体管理系统	58
	<i>Matthew Harmon</i>	
1.8.1	概述	58
1.8.2	实体消息	60
1.8.3	实体代码	61
1.8.4	类的代码	63
1.8.5	实体管理器	63
1.8.6	基于消息的游戏循环	65
1.8.7	开始: 消息类	65
1.8.8	从小处着手: 基本实体消息	66
1.8.9	游戏和环境消息	67

1.8.10	系统成长：一些高级消息	67
1.8.11	处理碰撞	69
1.8.12	扩展到多玩家	69
1.8.13	开发和调试消息	70
1.8.14	好处	70
1.8.15	光盘中的内容	71
1.8.16	总结	71
1.9	Windows 和 Xbox 平台上地址空间受控的动态数组	72
	<i>Matt Pritchard</i>	
1.9.1	传统的动态数组管理	72
1.9.2	深入观察	73
1.9.3	地址空间管理 != 存储管理	73
1.9.4	重新思考关于数组增大的问题	74
1.9.5	新的增长规则	74
1.9.6	使用地址空间受控的数组	75
1.9.7	结论	79
1.10	用临界阻尼实现慢入慢出的平滑	80
	<i>Thomas Lowe</i>	
1.10.1	可用的技术	80
1.10.2	阻尼弦与临界阻尼	82
1.10.3	实践	82
1.10.4	设置平滑速率的上限	84
1.10.5	结论	85
1.10.6	参考文献	85
1.11	一个易用的对象管理器	86
	<i>Natalya Tatarchuk</i>	
1.11.1	对象管理的传统做法	86
1.11.2	灵活的对象管理器	87
1.11.3	结论	91
1.11.4	参考文献	92
1.12	使用自定义的 RTTI 属性对对象进行流操作及编辑	93
	<i>Frederic My</i>	
1.12.1	扩展的 RTTI	93
1.12.2	属性	95
1.12.3	编辑属性	97
1.12.4	保存	99
1.12.5	载入	100
1.12.6	与旧版本文件的兼容性问题：类的描述	101
1.12.7	与旧版本文件的兼容性问题：匹配	102

1.12.8 “函数”属性	103
1.12.9 技巧和提示	103
1.12.10 思考	104
1.12.11 结论	104
1.12.12 参考文献	104
1.13 使用 XML 而不牺牲速度	106
<i>Mark T. Price</i>	
1.13.1 为什么要使用 XML 呢?	106
1.13.2 简单介绍 XDS Meta 格式	107
1.13.3 XDS 工具集	108
1.13.4 使用 XDS 工具集	109
1.13.5 整合	115
1.13.6 总结	115
1.13.7 参考文献	115

第 2 章 数学

简介	118
<i>Jonathan Blow</i>	
2.1 使用马其赛特旋转的 Zobrist 散列法	120
<i>Toby Jones</i>	
2.1.1 Zobrist 散列	120
2.1.2 实现 Zobrist 散列	121
2.1.3 马其赛特旋转 (Mersenne Twister)	122
2.1.4 马其赛特旋转的实现	123
2.1.5 结论	124
2.1.6 参考文献	124
2.2 抽取截锥体和 camera 信息	125
<i>Waldemar Celes</i>	
2.2.1 平面变换 (Plane Transformation)	125
2.2.2 抽取锥体信息	127
2.2.3 抽取 camera 信息	128
2.2.4 任意投影变换	130
2.2.5 实现	131
2.2.6 结论	132
2.2.7 参考文献	132
2.3 解决大型游戏世界坐标中的精度问题	133
<i>Peter Freese</i>	
2.3.1 问题描述	133
2.3.2 可能的解决方式	135

2.3.3	偏移位置	137
2.3.4	渲染流水线变化	140
2.3.5	对性能的思考	143
2.3.6	结论	143
2.3.7	参考文献	144
2.4	非均匀样条	145
<i>Thomas Lowe</i>		
2.4.1	样条的种类	145
2.4.2	三次样条的基础理论	146
2.4.3	圆形的非均匀样条	147
2.4.4	平滑非均匀样条	149
2.4.5	时控的非均匀样条	151
2.4.6	计算起始和最终节点速率	152
2.4.7	在样条上获取速率和加速度	153
2.4.8	优化	153
2.4.9	结论	153
2.4.10	参考文献	154
2.5	用协方差矩阵计算更贴切的包围对象	155
<i>Jim Van Verth</i>		
2.5.1	协方差矩阵	155
2.5.2	特征值和特征向量	158
2.5.3	计算协方差矩阵的特征向量	159
2.5.4	创建包围对象	159
2.5.5	结论	161
2.5.6	参考文献	162
2.6	应用于反向运动的雅可比转置方法	163
<i>Marco Spoerl</i>		
2.6.1	我们的测试环境	163
2.6.2	雅可比矩阵是什么?	164
2.6.3	雅可比转置矩阵简介	165
2.6.4	实现算法	166
2.6.5	结果和比较	168
2.6.6	结论	171
2.6.7	参考文献	171

第3章 物理

简介	174
<i>Graham Rhodes</i>	
3.1 死神的十指：战斗中的命中算法	176

<i>Roger Smith、Don Stoner</i>	
3.1.1 射击带状物 (Ribbon)	176
3.1.2 射击靶心	177
3.1.3 射击矩形	178
3.1.4 使用霰弹枪射击小目标	179
3.1.5 移动炮兵的攻击命中	179
3.1.6 死亡的 4 种主要形式	180
3.1.7 化学武器、火球及区域性魔法	182
3.1.8 弹片的楔入	182
3.1.9 攻击丛林	183
3.1.10 攻击有猎物分布的丛林	183
3.1.11 结论	184
3.1.12 参考文献	184
3.2 在低速 CPU 系统中交通工具的物理模拟	185
<i>Marcin Pancewicz、Paul Bragiel</i>	
3.2.1 技术的概要和前提假设	185
3.2.2 交通工具沿当前行驶方向上的加速及减速	186
3.2.3 方向控制	188
3.2.4 把所有要素结合起来	189
3.2.5 地形的影响	189
3.2.6 实现中遇到的问题	190
3.2.7 可以改进的地方	191
3.2.8 结论	192
3.3 编写基于 Verlet 积分方程的物理引擎	193
<i>Nick Porcino</i>	
3.3.1 关于物理引擎	193
3.3.2 刚体	194
3.3.3 积分器	194
3.3.4 物理引擎	196
3.3.5 针对特定平台的考虑	199
3.3.6 扩展引擎的功能	199
3.3.7 结论	200
3.3.8 参考文献	200
3.4 刚体动力学中的约束器	201
<i>Russ Smith</i>	
3.4.1 基本要点	201
3.4.2 约束器构造模块	202
3.4.3 创建有用的游戏约束器	205
3.4.4 光盘中的内容	209

3.4.5 结论 209

3.4.6 参考文献 209

3.5 在动力学模拟中的快速接触消除法 210

Ádám Moravánszky、Pierre Terdiman

3.5.1 减少接触 210

3.5.2 对预处理的详细分析 213

3.5.3 对接触的分组群的详细分析 214

3.5.4 对持续性的详细分析 217

3.5.5 结论 217

3.5.6 参考文献 218

3.6 互动水面 219

Jerry Tessendorf

3.6.1 线性的波浪 220

3.6.2 垂直导数操作符 221

3.6.3 波浪的传播 222

3.6.4 可以互动的障碍物及其发生源 224

3.6.5 环境波浪 225

3.6.6 网格的边界 225

3.6.7 表面张力 226

3.6.8 结论 226

3.6.9 参考文献 226

3.7 用多层物理模拟快速变形 228

Thomas Di Giacomo、Nadia Magnenat-Thalmann

3.7.1 基于物理的动画 LOD 及相关的工作 228

3.7.2 使用分层的质量块弹簧物理的快速变形 230

3.7.3 结论 235

3.7.4 参考文献 236

3.8 快速且稳定的形变之模态分析 237

James F. O'Brien

3.8.1 模式分解 239

3.8.2 模式的理解和丢弃 241

3.8.3 模态模拟 242

3.8.4 总结 244

3.8.5 结论 244

3.8.6 参考文献 245

第 4 章 人工智能

简介 248

Paul Tozour

4.1 第三人称视角摄像镜头的运动规则	250
<i>Jonathan Stone</i>	
4.1.1 Camera 定位及运动	250
4.1.2 Camera 与场景边界	253
4.1.3 Camera 遮断	256
4.1.4 简化场景	258
4.1.5 结论	258
4.1.6 参考文献	258
4.2 叙述战斗：利用 AI 增强动作游戏中的张力	260
<i>Borut Pfeifer</i>	
4.2.1 戏剧张力	260
4.2.2 系统概述	263
4.2.3 设计者的控制部分	263
4.2.4 难度计算	264
4.2.5 难度调节	265
4.2.6 系统评价	266
4.2.7 结论	267
4.2.8 参考文献	267
4.3 非玩家角色决策：处理随机问题	268
<i>Karén Pivazyan</i>	
4.3.1 概要	268
4.3.2 动态规划算法	269
4.3.3 代码	273
4.3.4 优化	275
4.3.5 DP 算法的其他应用	275
4.3.6 结论	276
4.3.7 参考文献	276
4.4 一个基于效用的面向对象决策架构	277
<i>John Hancock</i>	
4.4.1 决策树	278
4.4.2 基于对象的更好的体系结构	278
4.4.3 期望值	280
4.4.4 其他的决策准则	281
4.4.5 结论	282
4.4.6 参考文献	283
4.5 一个分布式推理投票架构	284
<i>John Hancock</i>	
4.5.1 分布式推理	284
4.5.2 操纵仲裁者 (Steering Arbiter) 范例	286

4.5.3	选择投票空间	289
4.5.4	结论	290
4.5.5	参考文献	291
4.6	吸引子和排斥子	292
<i>John M. Olsen</i>		
4.6.1	合力	292
4.6.2	引力曲线	293
4.6.3	吸引曲线的和	294
4.6.4	对应于特定配对的特定曲线	295
4.6.5	动态曲线	295
4.6.6	点、线、面	297
4.6.7	AI 控制的层次	298
4.6.8	动画系统的交互	298
4.6.9	移动 (Steering)	299
4.6.10	结论	299
4.6.11	参考文献	299
4.7	高级 RTS 游戏造墙算法	301
<i>Mario Grimaldi</i>		
4.7.1	算法	301
4.7.2	算法改进	302
4.7.3	输出链表的形式	306
4.7.4	结论	307
4.7.5	参考文献	307
4.8	利用可编程图形硬件处理人工神经网络	308
<i>Thomas Rolfes</i>		
4.8.1	CPU 与 GPU 系统架构	308
4.8.2	人工神经网络	309
4.8.3	实现	310
4.8.4	结论	311
4.8.5	参考文献	311

第 5 章 图形图像

简介	314
<i>Alex Vlachos</i>	
5.1 具有海报质量的屏幕截图	316
<i>Steve Rabin</i>	
5.1.1 提高分辨率	316
5.1.2 提升像素质量	318
5.1.3 使用一个磁盘均衡采样分布	320

5.1.4	为抗锯齿调整像素的采样宽度	321
5.1.5	增加分辨率同增加像素质量相结合	321
5.1.6	结论	323
5.1.7	参考文献	324
5.2	非封闭网络模型的 GPU 容积阴影构架	325
	<i>Warrick Buchanan</i>	
5.2.1	回到制图板	325
5.2.2	在顶点阴影中实现这项技术	326
5.2.3	需要注意的事项	329
5.2.4	结论	330
5.2.5	参考文献	330
5.3	透视阴影贴图	331
	<i>Marc Stamminger</i>	
5.3.1	引言	331
5.3.2	后透视空间	332
5.3.3	后透视空间中的光	334
5.3.4	透视阴影贴图	335
5.3.5	实现	338
5.3.6	结论	339
5.3.7	参考文献	340
5.4	结合使用深度和基于 ID 的阴影缓冲	341
	<i>Kurt Pelzer</i>	
5.4.1	已有的阴影映射技术	341
5.4.2	深度和基于 ID 的阴影缓冲	342
5.4.3	结合深度和 ID 缓冲	343
5.4.4	组合的阴影缓冲概述	344
5.4.5	第一次：从光照的视点渲染	345
5.4.6	第二次：阴影检测	347
5.4.7	在 DX9 2.0 级的阴影中的实现	351
5.4.8	结论	353
5.4.9	参考文献	353
5.5	在场景中投射静态阴影	355
	<i>Alex Vlachos</i>	
5.5.1	前期工作	355
5.5.2	光束基本知识	355
5.5.3	高级算法	356
5.5.4	T 形连接	357
5.5.5	网格模型最优算法	358
5.5.6	实现细节	359

5.5.7 阴影中的动态物体 360

5.5.8 结果 360

5.5.9 结论 361

5.5.10 参考文献 361

5.6 为阴影体和优化的网格模型调整实时光照 362

Alex Vlachos、Chris Oat

5.6.1 光照问题 362

5.6.2 在面法线上操作 362

5.6.3 调整漫射光照 364

5.6.4 结论 366

5.6.5 参考文献 366

5.7 实时半调色法：快速而简单的样式化阴影 367

Bert Freudenberg、Maic Masuch、Thomas Strothotte

5.7.1 引言 367

5.7.2 原理 368

5.7.3 实例的实现 371

5.7.4 结论 372

5.7.5 参考文献 372

5.8 在 3D 模型中应用团队色的各种技术 373

Greg Seegert

5.8.1 什么是团队色? 373

5.8.2 团队色的算法 373

5.8.3 一个实际的例子 378

5.8.4 光盘中的内容 379

5.8.5 结论 379

5.9 快速的棕褐色色调转换 380

Marwan Y. Ansari

5.9.1 背景 380

5.9.2 常规的方法 380

5.9.3 优化 381

5.9.4 结论 382

5.9.5 参考文献 382

5.10 使用场景亮度采样实现动态的 Gamma 383

Michael Dougherty、Dave McCoy

5.10.1 光照系数 383

5.10.2 有限的动态范围 383

5.10.3 图像的优化 384

5.10.4 易变的光灵敏度 385

5.10.5 转换 386

5.10.6	算法	388
5.10.7	结论	392
5.11	热和薄雾的后处理效果	393
	<i>Chris Oat、Natalya Tatarchuk</i>	
5.11.1	热和闪光的薄雾	393
5.11.2	高级算法	393
5.11.3	计算失真值	394
5.11.4	失真值的解释	397
5.11.5	结论	400
5.11.6	参考文献	400
5.12	用四元数的硬件蒙皮	401
	<i>Jim Hejl</i>	
5.12.1	蒙皮的概念	402
5.12.2	四元数参数化	404
5.12.3	硬件实现	405
5.12.4	结论	407
5.12.5	参考文献	407
5.13	动作捕捉数据的压缩	409
	<i>Søren Hannibal</i>	
5.13.1	处理的计划	409
5.13.2	组织数据通道	410
5.13.3	减少已储存的键的数量	410
5.13.4	包装剩余的键	412
5.13.5	运行时解压缩	412
5.13.6	未来的改进	413
5.13.7	结论	413
5.13.8	参考文献	413
5.14	基于骨骼的有关节的 3D 角色的快速碰撞检测	414
	<i>Oliver Heim、Carl S. Marshall、Adam Lake</i>	
5.14.1	碰撞检测与碰撞分解	414
5.14.2	术语	414
5.14.3	将碰撞检测集成到 3D 游戏引擎中	415
5.14.4	基于骨骼的快速碰撞检测算法	416
5.14.5	结论	423
5.14.6	感谢	423
5.14.7	参考文献	423
5.15	使用地平线进行地形的遮挡剔除	424
	<i>Glenn Fiedler</i>	
5.15.1	引言	424

5.15.2	地平线剔除基础	425
5.15.3	蛮力地平线剔除	426
5.15.4	近似值	426
5.15.5	近似地平线直线	427
5.15.6	一个更好的近似值	427
5.15.7	最小二次方线	428
5.15.8	将它放入到第三维中	429
5.15.9	最小二次方平面	430
5.15.10	用近似值的地平线剔除	431
5.15.11	被地形遮挡的对象	432
5.15.12	使它成为动态的	432
5.15.13	未来的方向	433
5.15.14	结论	433
5.15.15	参考文献	433

第6章 网络和多人游戏

简介	436
<i>Pete Isensee</i>	
6.1 设计与开发游戏大厅	437
<i>Shekhar Dhupelia</i>	
6.1.1 状态—事件系统的设计	437
6.1.2 探讨大厅的子系统	438
6.1.3 高级大厅子系统	439
6.1.4 结论	441
6.1.5 参考文献	442
6.2 支持成千上万个客户端的服务器	443
<i>Adam Martin</i>	
6.2.1 服务器设计中的门槛	443
6.2.2 问题	444
6.2.3 主要技术	446
6.2.4 服务器设计	451
6.2.5 结论	452
6.2.6 参考文献	452
6.3 大型多人游戏状态的有效存储	454
<i>Justine Quimby</i>	
6.3.1 MMP 的问题	454
6.3.2 Qualities 理论	455
6.3.3 Qualities API	456
6.3.4 使用 Qualities 的好处	459

6.3.5 结论 459

6.3.6 参考文献 460

6.4 在客户/服务器环境下运用并行状态机..... 461

 Jay Lee

6.4.1 独立状态 461

6.4.2 角色状态管理器 463

6.4.3 使用 CharacterStateMgr..... 464

6.4.4 保持客户端和服务端同步 464

6.4.5 状态依赖的子系统 466

6.4.6 结论 467

6.4.7 参考文献 467

6.5 位打包：一种网络压缩技术 468

 Pete Isensee

6.5.1 一个实例 468

6.5.2 难点 469

6.5.3 位打包 469

6.5.4 用于可打包数据类型的通用接口 471

6.5.5 用于可打包数据类型的具体接口 471

6.5.6 编解码器 472

6.5.7 评价折衷 473

6.5.8 改进 473

6.5.9 结论 473

6.5.10 参考文献 474

6.6 多服务器网络游戏的时间和同步管理..... 475

 石卫东 (Larry Shi)、Tao Zhang

6.6.1 为什么需要时间和同步管理 475

6.6.2 时钟同步 475

6.6.3 同步和响应 476

6.6.4 用多时间管理来一石二鸟地实现同步和响应 476

6.6.5 实现 476

6.6.6 何时应使用多时管理 482

6.6.7 总结 482

6.6.8 致谢 482

6.6.9 参考文献 482

第 7 章 音频

简介 486

 Eddie Edwards

7.1 OpenAL 简介 487

<i>Joe Valenzuela</i>	
7.1.1	OpenAL API..... 487
7.1.2	有关 OpenAL 的实现..... 493
7.1.3	实现一致性指南..... 495
7.1.4	未来 OpenAL 的发展蓝图..... 496
7.1.5	总结..... 496
7.1.6	参考文献..... 496
7.2	简单的实时 Lip-Synching 系统..... 497
<i>Jake Simpson</i>	
7.2.1	实现..... 497
7.2.2	动画方面需要注意的事项..... 498
7.2.3	声音音量的水印标记..... 499
7.2.4	注意..... 500
7.2.5	总结..... 500
7.3	动态变量和音频编程..... 501
<i>James Boer</i>	
7.3.1	动态变量是什么?..... 501
7.3.2	动态变量类..... 501
7.3.3	在音频编程中使用动态变量..... 503
7.3.4	其他改进..... 506
7.3.5	结论..... 506
7.3.6	参考文献..... 506
7.4	创建一个音频脚本系统..... 507
<i>Borut Pfeifer</i>	
7.4.1	游戏中音频的类别..... 508
7.4.2	工具..... 510
7.4.3	基于 XML 的音频标记库..... 510
7.4.4	脚本系统组件..... 512
7.4.5	进一步的工作..... 515
7.4.6	总结..... 515
7.4.7	参考文献..... 515
7.5	使用 EAX 和 ZoomFX API 的环境音效解决方案..... 516
<i>Scott Velasquez</i>	
7.5.1	什么是环境音效..... 516
7.5.2	音频引擎的系统要求..... 517
7.5.3	潜在可听集 (PAS, Potentially Audible Set)..... 518
7.5.4	EAX 介绍..... 520
7.5.5	总结..... 528
7.5.6	参考文献..... 528

7.6 在游戏的物理引擎中控制实时声音 529

Frank Luchs

 7.6.1 游戏引擎 529

 7.6.2 混合声音合成 531

 7.6.3 可听见对象的属性 532

 7.6.4 对象形状的影响 532

 7.6.5 对象材质的影响 533

 7.6.6 撞击和碰撞 533

 7.6.7 演示 533

 7.6.8 总结 534

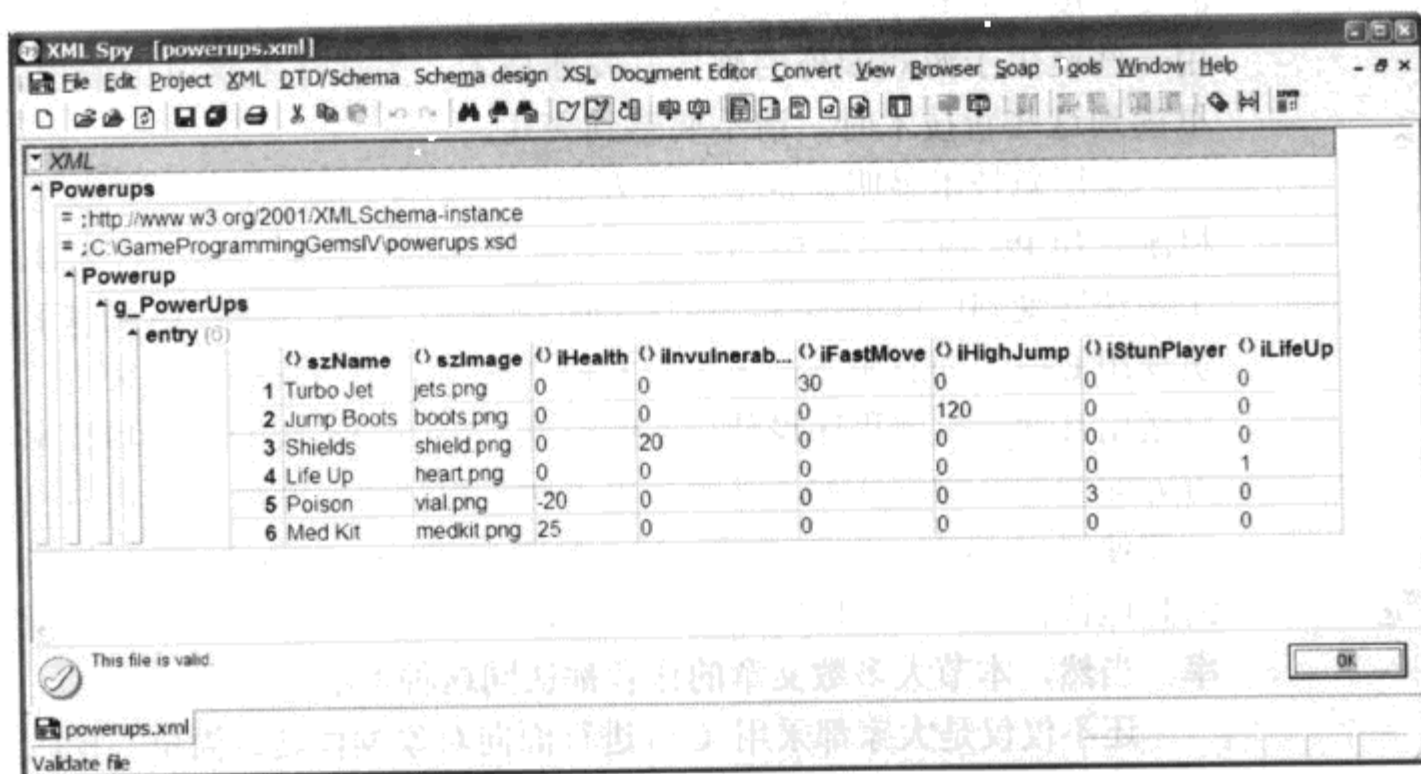
 7.6.9 参考文献 534

附录 536



第 1 章

通用编程



简介

作者: Chris Corry, LucasArts

E-mail: chris@thecorrys.com

译者: 沙鹰

随便找来两个游戏程序员，再找一间小办公室让他们呆在一起，你很快就会发现他们对很多事情意见相左。干游戏这一行的程序员们，通常都有着很强的个性，有时候甚至可以说是富有叛逆精神。不过，如果说有些事情，游戏程序员们能够罕有地达成一致意见，下面这句话一定是其中之一：游戏开发比从前容易那是胡说。事实上，游戏程序员们多半会告诉你，不论是电脑游戏也好，电视游戏也好，游戏总之是一年比一年难做了。的确，我们今日面临的最大挑战之一就是控制游戏软件的复杂度。因此不可避免地，我们中的许多人已经开始在游戏开发中采用一些诞生于游戏圈外的软件开发新技术。当你在本章讨论编程的文章里读到这些新技术时，请不要感到突兀。

采用新技术最明显也最广为人知的一个例子就是 C++ 语言在家用游戏机游戏和 PC 游戏开发中的运用。还在数年前，用 C++ 开发游戏机游戏这个想法还受到广泛的怀疑，但这个怀疑现在已经没有了。越来越多的程序员使用 C++，而且不是简单地将 C++ 作为一个“更好的 C”来用，而是充分运用其面向对象程序设计语言的特性，对我们游戏的体系和技术设计的方方面面都有深远影响。当然，出于对于程序效率的执著，我们也可能要按需使用 C 语言、汇编语言，甚至是微代码（microcode），但我们越来越清楚地认识到，合理地用 C++ 进行面向对象程序设计，并非一定要牺牲效率。当然，本节大多数文章的作者都认同这种说法。

还不仅仅是大家都采用 C++ 进行面向对象编程这么简单。模版元编程（template meta-programming）、STL、UML、XML、设计模式等等所有这些“与游戏无关”的技术或方法学，哪怕是再保守的工作室也至少会利用一二。在本节里，你会看到许多文章都参考了这些工具或技术。当大家都适应它们的那天，它们必定会在你的编程“百宝箱”里永久地占据一席之地。

在我们的行业里，有着如此多新奇、独特和使人兴奋的东西。我们不单是在设计和制造娱乐产品，而且在游戏开发的过程中，我们多的就是接触那些绝对不会在他处接触到的硬件和软件的机会。在这个行业里，每天都有技术革新发生，也难怪有时候我们这些游戏开发者们，会对“外部的”技术感到怀疑和无动于衷，毕竟那些新东西还没有经过我们的实际使用与检验。



然而，我们还是有理由不认同这样保守的做法。游戏业外的软件开发者们，常常面临着一些与我们天天忙于应付的问题极为相近的问题。应如何设计灵活的数据格式，使其在长达2~3年的项目开发周期里都很容易改进？应如何快速有效地向内存载入大量的数据？应如何消除设计决定和编程决定之间的耦合性，使我们能够设计出方便的工具，最终将工作重心放在高质量的内容开发而不是其他东西上？须知并不是只有游戏业遇到了这些问题，将目光放远，一定会有收获。不要怕，试着做些与以往不同的事。买一本关于企业软件开发的专门杂志，读一本专注 UML 应用、Java 或 SQL 数据库的期刊，你会收获惊喜。



1.1 调试游戏程序的学问

作者: Steve Rabin, Nintendo of America Inc.

E-mail: steve@aiwisdom.com

译者: 沙鹰

审校: 万太平

调试游戏程序, 和调试任何其他软件的代码一样, 都可能是非常艰巨的任务。一般说来, 有经验的程序员能迅速地识别并纠正哪怕是最难的 bug; 但是对于新手而言, 改 bug 则很可能是件难以处理的且容易使人灰心丧气的任务。更糟的是, 当你初步着手开始寻找 bug 的根源时, 永远也不会知道究竟要花费多长时间才能找到。此时不必慌张, 要像个训练有素的程序员, 集中精力寻找 bug。一旦你消化了本文介绍的技巧和知识, 你将能够击退最“凶猛”的 bug, 重获对游戏的控制。

运用本文描述的五步调试法, 困难的调试过程可变得简单一些。训练有素地运用该方法, 将确保你花费最少的时间在寻找和定位每一个 bug 上。在你着手对付一些有难度的 bug 时, 牢记一些专家技巧也很重要, 因此本文也收集了一些有价值的、经过时间考验的技巧。然后本文还列出了一些有难度的调试情境, 解释了当遇到一些特定的 bug 模式时应当做些什么。因为好的工具对于调试任何游戏都很重要, 本文还将讨论一些特定的工具, 你可将这些工具嵌入你的游戏中, 从而帮助调试一些游戏编程所独有的调试情形。最后我们将回顾一些在前期预防 bug 的简单技术。

1.1.1 五步调试法

老练的程序员们具有一种超能力, 能够迅速地、驾轻就熟地捕捉到即使是最不可思议的 bug。他们总是神奇地、近乎直觉地知道错误源自何方, 这一点实在令人敬畏。他们之所以显得天才, 除了因为拥有丰富的经验外, 还因为他们对于勘探和减少需排查的可能原因的方法训练有素、融会贯通。下面给出的五步调试法旨在重现他们所熟练掌握的技能, 助你在跟踪 bug 的问题上形成一种有系统的、且注意力集中的风格。

1.1.2 第一步: 始终如一地重现问题

不论是什么 bug, 重要的是, 你应当了解如何能够始终如一地重现它。

试图纠正一个随机出现的 bug 常会使人感到挫败，而且通常不过是浪费时间。事实是，几乎所有的 bug 都会在特定的情境下可靠地重现，因此发现这个情景和规律就成为你的或贵公司测试部同仁的工作。

让我们举一个假想的游戏 bug 为例，在测试员报告里写道：“有时候，游戏会在玩家杀死敌人时死机 (crash)。”很不幸，像这样的 bug 报告过于含糊，而且由于这个问题看上去不是百分之百会出现的，多数时候玩家仍可以正常地摧毁敌人。因此当游戏 crash 时，必然还有一些其他相关因素。

对于不容易重现的 bug，理想情形是创建一系列“重现步骤 (repro step)”，说明每次应怎样才能重现 bug。例如，下面的步骤极大地改善了之前的 bug 报告。

重现步骤。

- (1) 开始单人游戏。
- (2) 选择在第 44 号地图上进行 Skirmish 也就是多人练习模式的游戏。
- (3) 找到敌人营地。
- (4) 在一定的距离开外，使用投射类武器 (projectile weapon) 攻击在营地里的敌人。
- (5) 结果：90% 的时候游戏死机。

显然，重现步骤是一种很好的方法，测试人员借此帮助其他人重现 bug。不过，精简可能导致 bug 发生的事件链 (chain of event) 的过程也是至关重要的。其原因有三：第一，对当时 bug 为何发生提供了有价值的线索；第二，提供了一种比较系统地测试 bug 是否已被彻底改正的方法；第三，可用于回归测试，确保 bug 不再卷土重来。

尽管这里的信息没有告诉我们 bug 的直接诱因，但它使我们能够始终重现 bug。一旦你确定了 bug 发生的环境，你就可以进行下一步骤，开始搜集有用的线索。

1.1.3 第二步：搜集线索

现在你能够可靠地使 bug 重现，下一步请你戴上侦探的鸭舌帽并搜集线索。每个蛛丝马迹都是排除一个可能原因并缩短疑点列表的机会。有了足够的线索，bug 的发源地会变得明显。因此为了明了每条线索并理解其潜台词，付出的努力是值得的。

不过有一点要注意，你应当总是在心里质疑每一条已发现的线索，是不是误导或不正确的。举例来说，我们被告知某个 bug 总发生在爆炸之后。尽管这可能是一条非常重要的线索，但它仍然可能是一个虚假的误导。要时刻准备放弃那些与收集来的信息冲突的线索。

还是以上面的 bug 报告为例，我们了解到游戏的 crash 发生在玩家使用投射类武器攻击某个特定的敌人营地的时候。究竟关于投射类武器和从远处攻击这两者，有什么特别之处？这是需要深思的重点，但也不要耗费太多时间思考。亲临其境，观察错误究竟是如何发生的，因为我们需要获取更多的确凿证据，而流连于表面的线索是获得实际证据最无效的方式。

在本例中，当我们进入游戏，并实际观察错误的发生时，我们会发现游戏死机发生在一个“箭”对象里，错误的症状是一个无效指针。进一步的检查显示，该指针本来是应当指向那个发射此箭的角色的。在此情况下，这支箭原本要向其发射者报告它击中了某个敌人，使发射者为该次成功的攻击获得一定的经验值。但尽管看上去找到了原因所在，我们对真实的潜在原因仍然一无所知。我们必须首先找出是什么扰乱了这个指针。

1.1.4 第三步：查明错误的源头

当你认为收集到的线索已经够多时，就到了专注于搜索和查明错误源头的时候了。有两个主要方法，第一个方法是先提出关于 bug 发生原因的假设，接着对该假设进行验证（或证明它不正确）；第二个方法是较为系统的分而治之法。

方法 1：假设法

搜集了足够的线索，你会开始怀疑是些什么事情导致了 bug 发生。这就是你的假设（hypothesis）。当你能够在心里清楚地陈述假设，你就可以开始设计一些能验证该假设，或反证证明该假设不正确的测试用例。

在我们的例子里，通过测试得出了以下线索和关于游戏设计的信息。

- 当一支箭射出的时候，该箭被赋予一个指向射箭人的指针。
- 当一支箭射中某个敌人的时候，将奖励送给射箭人。
- 游戏死机发生在一支箭试图通过一个无效指针向射箭人传回奖励。

我们的第一个假设可能是，指针的值在箭的飞行途中被损坏。基于此种假设，我们开始设计测试，并搜集数据来支持或推翻此原因。例如我们可以让每一支箭都将射箭人的指针注册到同一个备份区域。当我们又捕捉到 crash 时，可以检查备份下来的数据，看无效指针的值是否与这支箭在被射出的时候所赋予的值相同。

不幸的是在我们所举的例子里，最后发现这条假设是不正确的。备份的指针和导致游戏死机的指针具有相同的值。这样一来，我们就面临着一个抉择。是再提一个假设并进行验证，还是重头寻找更多的线索？现在让我们试着再提一条假设。

如果箭的发射人指针从没有被破坏（新线索），或许从箭射出到箭射中敌人的这段时间里，这个发射人被删除了。为了检查这点，让我们记录下敌人营地中死亡的每个角色的指针。当 crash 发生时，我们可以将出错指针和死亡并从内存中删除的敌人列表进行比较。这样，很快就证实原因正是如此。射箭人死时，箭还在飞行途中。

方法 2：分治法

两个假设使我们找出了 bug，同时也表现了分而治之的概念。我们知道指针的值无效，但我们不知道它是因为值被修改过而损坏，还是因为在更早些的时候这个指针就已经无效。通过测试第一个假设，我们排除了两个可能性中的一个。像歇洛克·福尔摩斯（Sherlock Holmes）曾说过的：“……当你排除了不可能的情况后，其余的情况，尽管多么不可能，却必定是真实的。”^①

有人将分而治之的方法简单形容为，确定故障发生的时刻，并从输入开始回溯来发现错误。比如有一个并不会造成死机的 bug，在某个时刻发生的初始错误将影响层层传递，最终导致故障发生。确定初始错误通常通过在所有输入分支上设置（有条件或无条件的）断点（breakpoint）来进行，直到找到那个不能正常输出——也就是导致 bug 的输入。

^① 译者注：引自柯南·道尔《绿玉皇冠案》。

当从故障发生的时刻开始回溯，你在局部变量和栈里面的上级函数中寻找任何异常。对于死机 bug 来说，通常会试图寻找一个空值（NULL）或极大的数字值。如果是关于浮点数的 bug，在栈上寻找 NAN 或极大的数字。

无论是对问题进行有根据的推测、检验假设，还是有系统地搜捕肇事代码，最终你会找到问题所在。在这个过程中你要相信自己，并保持清醒。本文接下来的部分将详细讨论一些可用于这步骤的专门技术。

1.1.5 第四步：纠正问题

当我们发现 bug 的真正根源，接下来要做的便是提出和实现一个解决方案。然而所做的修改必须适合项目所处的阶段。例如，在开发的后期，通常不能只为了纠正一个 bug，就修改底部的数据结构或程序体系结构。参照开发工作所处的阶段，主程序员或系统架构师将决定应当进行何种类型的修改。在关键的时刻，个别工程师（初级或中级）常常做出不好的决定，因为他们没有全盘考虑。

此外，需要特别注意的是，在理想情况下，代码的编写者应当负责修改自己代码里的 bug。不过如果必须修改别人的代码，你至少应当在进行修改前和原作者进行讨论。通过讨论，你将了解到一些情况，例如在以往对于类似的问题是怎么处理的，如果实施你提议的方案可能会造成什么影响等。总之，在未彻底理解由别人编写的代码的上下文前，急于进行修改是非常危险的。

继续讨论我们的例子，死机源于一个指向了一个不复存在的对象的无效指针。对此类问题模式的一个好的解决方案是使用一层间接引用，使 crash 不再发生。通常，正是因为这个理由，游戏使用对象的句柄而不是使用直接指针。这将是一个合理的修改。

但是，如果游戏项目因为某个里程碑、或一个重要的演示版交付日迫在眉睫，而需要快速完成修改，你可能会倾向于对现有的特殊情况实施一个较为直接的修改方案（例如让射箭者在自身被删除的时候使其射出的箭中关于自身的指针失效）。如果在程序里打上了这一类的快速补丁（quick hack），你要记得将有关的注释文档化，以使其在这截止日期后被重新评估。开发中这样的情况屡见不鲜：快速补丁被人们遗忘，而在几个月后才造成了难于发现和解决的麻烦。

虽然看上去我们发现了 bug 并且确定了一种修改方式（使用句柄而非指针），但探索其他可能造成同样问题出现的途径是很关键的。这虽然需要额外的时间，但是为了确保 bug 从根本上被消灭，而非只是消除了 bug 的一种表现形式，这种努力是值得的。在我们的例子中，可能其他类型的投射类武器同样会造成游戏死机，但其他非武器对象的关系、甚至角色之间的关系也会受到同一个设计缺陷的影响。应找出所有这些相关的场合，使你的修改方案针对的是问题的核心，而非仅仅是问题的某一种表现。

1.1.6 第五步：对所作的修改进行测试

解决方案实施后，还必须进行测试以确认它的确修补了错误。第一步要确保先前有效的重现步骤不会导致 bug 重现。通常应当让 bug 修改者以外的其他人，例如测试员，独立地确认 bug 是否被修复。

第二步还要确保没有新的 bug 被引入游戏。你应当让游戏运行一段相当长的时间，确保所作的修改没有影响其他部分。这是非常重要的，因为很多时候，尤其是在项目开发周期接近尾声的时候，为修改 bug 所作的改动，会导致其他系统出错。在项目的后期，你还应当让主程序员或其他开发者来检测每一个修改，这额外的可靠性检验要保证新的修改不会对版本有负面影响。

1.1.7 高级调试技巧

如果你遵循以上所述的基本调试步骤，你应当能找到并修复大多数 bug。不过在你尝试提出假设、验证/否决一个候选的原因、或者尝试找出出错位置的时候，或许你会愿意考虑采取下列的技巧。

1. 分析你的假设

调试程序的时候，保持思路开扩是很重要的，而且不要作太多假设。如果你假设某些貌似简单的东西总是正确的，你可能就会过早地缩小了搜索范围，从而完全错过了找出真相的机会。举例来说，不要总是想当然地认为你正在使用最新的软件或程序库。检验你的假设是否正确常常是值得的。

2. 将交互和干扰最小化

有时，多个系统之间会以某种方式交互，这会使调试复杂化。试试看关闭那些你认为和问题无关的子系统（例如，关闭声音子系统），从而将系统之间的交互降到最低限度。有时候这有助于识别问题，因为原因可能就在你关闭的系统中，这样你就知道接下来该看哪里。

3. 将随机性最小化

通常，bug 之所以难于重现，要归咎于从帧速率和实际随机数等方面引入的可变性。如果你的游戏没有采取固定的帧速率，试试看将“在每帧内流逝的时间”锁定为常量。至于随机数，可以关闭随机数发生器，或给它固定的常数作为随机发生种子，这样每次运行都会得到同样的序列。不幸的是，玩家会给游戏带来无法控制的显著的随机性。如果连这玩家带来的随机性也必须得到控制，请考虑将玩家的输入记录下来，从而能以可预料的方式将输入记录直接送入游戏 [Dawson01]。

4. 将复杂的计算拆分成几步进行

若某行代码含有大量计算，或许将这行拆分为多个步骤会有助于识别问题。例如，可能其中的某小段计算产生了类型转换错误，或某个函数并未返回你期望它返回的值，或运算进行的顺序并不是你所想的那样。这也使你能够检查每一步中间过程的计算。

5. 检查边界条件

几乎我们每一个人都曾被经典的“差一错误”（off-by-one）问题折磨过。要检查算法的边界条件，特别是在循环结构中。

6. 分解并行计算

如果你怀疑程序里的竞争条件 (race condition, 不同的执行顺序会产生不同的结果), 试试看将代码改写为串行的, 然后检查 bug 是否消失。在线程中, 增加额外的延迟, 观察问题是否也随之变化。问题范围能缩小——若你能够确定问题是竞争条件, 并通过试验将问题孤立出来。

7. 充分利用调试器提供的工具

明白和懂得如何使用条件断点、内存 watch、寄存器 watch、栈, 以及汇编级/混合调试。工具能帮你寻找线索和确凿的证据, 这是识别 bug 的关键。

8. 检查新近改动的代码

调试也可以通过源代码版本控制来进行, 这真是一个令人惊讶的方法。如果你清楚地记得在某个日期前程序还是工作的, 但是从某天开始就失灵了, 你就可以专注于期间改动过的代码, 从而较快地找到引入缺陷的代码段。至少, 也可以将搜索范围缩小至某个特定子系统, 或某几个文件。

另一个利用版本控制的方法是生成游戏在 bug 出现之前的一个版本。当你看不清问题的时候这尤其有用。将新老版本分别在调试器中运行, 将值互相比对, 你就可能找出问题的关键所在。

9. 向其他人解释 bug

常常在你向他人解释 bug 的时候, 你会追忆起一些步骤, 并意识到一些遗漏或忘记检查的地方。与其他的程序员交流的益处还在于他们可能会精辟地提出别样的值得检验的假设。不要低估和他人交谈的作用, 也永远不要羞于征求他人的建议。你团队中的同事既是你的伙伴, 也是你与最有难度的 bug 战斗时最精良的武器之一。

10. 和同事一起调试

这通常是很合算的, 因为每个人在对付 bug 上都有自己的独门经验和策略。你也能学到新的技术, 学会从未尝试过的角度入手处理 bug。让别人看着你进行调试, 这可能是你追捕 bug 最有效的方法之一。

11. 暂时放下问题

有的时候, 你已经如此接近问题, 以至于无法再清楚全面地看待它。试试看改变一下环境, 出门闲逛一下。当你放松, 再回到问题上, 你可能会会有新的认识。有时候, 当你决定让自己休息一下时, 你的心里下意识地还在思考问题, 过后答案就自然浮现了。

12. 寻求外部的帮助

获得帮助有多种很好的途径。如果是在开发视频游戏, 那么每家游戏机制造商都有一整班的人, 他们将在你遇到麻烦的时候帮助你。去了解他们的联系方式吧。三大游戏机制造商现在都提供电话支持、电子邮件支持和开发者互相帮助的新闻讨论组。

1.1.8 困难的调试情景和模式

消灭 bug 常有模式可循。在艰苦的调试情景中，模式是关键。在此经验起了很大作用。如果你曾经见过某个模式，你就有可能迅速地找出 bug 所在。希望下列情景和模式能给你指明一些方向。

1. bug 仅在发布版里出现，调试版则正常

通常，bug 只出现于发布版（release build）中意味着这是数据未初始化，或与代码优化有关的 bug。一般来说，即使你没有特地编写进行初始化的代码，调试版（debug build）也会自动将变量初始化为零。而这隐式初始化在发布版中是不存在的，因而出现了 bug。

找出原因的另一个策略是：在调试版里，慢慢地逐一打开优化开关。对每一点优化都进行测试，你可以找到罪魁祸首。例如，在调试版里，函数一般都不是内联的。但在优化后有些函数自动进行了内联，有时某个 bug 就这样发作了。

还有一点值得注意的是，在发布版中也可以打开调试符号（debug symbol）。这使得在一定程度上（虽然一般并不会）对优化过的代码进行调试成为可能，你甚至可以让一部分调试系统保持开启。举例来说，你可以让你的异常处理函数在崩溃的现场执行一个全面的堆栈回溯（这需要符号）。这是非常有用的，因为当测试员必须运行优化过的游戏版本的时候，你还是可以回溯程序崩溃。

2. 在做了一些无害的改动后，bug 不见了

如果 bug 在一些完全无关的改动（例如添加了一行无害的代码）后不见了，那么这就像是一个时序问题，或内存覆盖问题。尽管表面上 bug 已经消失了，但是实际上可能只是转移到了代码的另一个部分。不要错过这个找出 bug 的机会。Bug 就在那儿，将来迟早有一天它肯定会不知不觉地、狡猾地害你。

3. 确实具有间歇性的问题

像前面提过的那样，许多问题会在合适的环境下稳定地重现。但如果你无法控制环境，那就必须趁问题抬起它丑陋的小脑袋时抓住问题。这里的关键是，在捕捉到问题的时候要记下尽可能多的信息，以便以后可在必要时检查。机会可不是很多的，因此要充分利用每一次出错的机会。还有一个有效的技巧就是将程序出错时收集得到的数据和程序正常时收集的数据进行比较，发现其中差异。

4. 无法解释的行为

有时当你在单步执行代码的时候，却发现变量自说自话地被修改了。这种真正怪异的现象通常表示系统或调试器失去了同步。解决方案是试试看“加快清除缓存的频率”，使系统重获同步。

感谢 Scott Bilas 为清除缓存归纳出如下的“四重”方针。

- 重试（retry）：清除游戏的当前状态再运行。

- **重建 (rebuild)**: 删除已编译过的中间对象, 并进行彻底的版本重建。
- **重启 (reboot)**: 通过硬复位, 将你机器里的内存擦除。
- **重装 (reinstall)**: 通过重装, 恢复你的工具和操作系统中的文件和设置。

在这“四重”里, “重建”是最重要的。有时候, 编译器不能正确地识别代码间的依赖关系, 导致受牵连的代码不能通过编译。症状常常是不可思议的怪异。一次彻底的重建有时就能解决问题。

处理这些无法解释的行为的时候, 一定要预先猜测调试器会给出何种结果。通过 `printf` 函数输出并检验变量的实际值, 因为调试器有时候会被迷惑, 而无法准确地反映真实的值。

5. 编译器内部错误

偶尔你会碰到这种情况, 编译器承认它无法理解你的代码, 从而抛出一个编译器内部错误 (`internal compiler error`)。这些错误可能显示在代码中存在合法性问题, 也可能根本是编译器软件自身的问题 (例如, 超出了内存上限, 或无法处理你如同天书一般的模版代码)。遇到编译器内部错误的时候, 建议执行如下步骤。

- (1) 进行完整的版本重建。
- (2) 重启电脑, 再进行一次完整的版本重建。
- (3) 检查是否正在使用最新版本的编译器。
- (4) 检查任何正在使用的库是否是最新版本。
- (5) 试验同样的代码是否能在其他电脑上通过编译。

如果这些步骤不能解决问题, 试试确定究竟是那段代码引起了错误。如果可能的话, 用分治法减少编译到的代码, 直至编译器内部错误消失。当故障的位置被确定后, 检视这段代码并保证它看上去没错 (最好能多请几个人读它)。如果代码看上去的确合理, 下一步试着重新组织一下代码, 希望编译器能报告出更有意义的错误信息。最后你还可以尝试用旧版本的编译器来编译。很可能在最新版的编译器里存在 `bug`, 而使用旧版本的编译器就能顺利完成编译。

如果这些办法都不奏效, 试试在网上搜索相似的问题。如果还是没有用, 向编译器的制造商寻求额外的帮助。

6. 当你怀疑问题不是出在自己的代码里

不像话, 应该总是怀疑自己的代码! 不过, 如果你确信不是你们自己代码的问题, 最好的办法是到网站上寻找所使用的函数库或编译器的更新补丁。详细阅读其 `readme` 文件, 或者在网上搜索关于此函数库或编译器的已知问题。很多时候, 其他的人也碰到了相似的问题, 解决办法或补丁也已经有了。

不过, 你发现的 `bug` 来自他人提供的函数库, 或来自有故障的硬件 (碰巧你是第一个发现它的人) 的几率不大。虽然不太可能, 但有时还是会发生的。最快的解决方法是编写一小段例程将问题隔离开来。然后你可以把这段程序 E-mail 给函数库的作者, 或硬件生产商, 以便他们进一步就此问题进行调查。如果这真是其他人造成的 `bug`, 由于你的帮助, 他人可以快速识别和重现问题, 从而 `bug` 将以最快速度得到改正。

1.1.9 理解底层系统

有时为了找到一些难度很高的 bug，你必须了解底层系统。仅仅通晓 C 或 C++ 还远远不够。为了成为一个优秀的程序员，你必须懂得编译器是如何实现较高层次的概念，必须懂汇编语言，还必须了解硬件的细节（尤其是对游戏机游戏开发而言）。虽然认为高级语言掩盖了所有的复杂性并没有错，但是事实是当系统崩溃时你会感觉手足无措，除非你理解了抽象层之下的底层系统。若要进一步讨论高层抽象会如何造成隐患，请参见“The Law of Leaky Abstractions”[Spolsky02]。

那么，有哪些底层细节需要了解呢？就游戏而言，你应当了解如下事项：

- 了解编译器实现代码的原理。熟悉继承、虚函数调用、调用约定、异常是如何实现的。懂得编译器如何分配内存和处理内存对齐。
- 了解你所使用的硬件的细节。例如，懂得与某个特定硬件的高速缓存有关的问题（缓存中的数据何时会和主存储器中不同）、内存对齐的限制、字节顺序（endianness，高位还是低位字节在前）、栈的大小、类型的大小（如整型 int、长整型 long、布尔型 bool）。
- 了解汇编语言的工作原理，能够阅读汇编代码。这在调试器无法跟踪源代码时，例如在优化后的版本里查找问题时，会很有帮助。

如不能牢牢掌握这些知识，在对付真正困难的 bug 的时候，你的致命弱点就会暴露出来。所以必须理解底层的系统，熟悉其规则。

1.1.10 增加有助于调试的基础设施

没有合适的工具帮助，在真空中调试程序必定会很费劲。解决办法是走另一个极端，直接将好的调试工具整合到游戏里。下列工具能极大地帮助修理 bug。

1. 允许在运行中修改游戏变量

调试和重现 bug 时，在运行中修改游戏变量值的功能是非常有用的。实现此功能的经典界面是通过游戏中的一个调试命令行接口（CLI，Command-Line Interface）用键盘修改变量。按下某个键后，调试信息覆盖显示在游戏屏幕上，提示你用键盘进行输入。例如，当你想把游戏里的天气改成狂风暴雨，你可以在提示下输入“weather stormy”。此类界面在调节和检查变量的值或特定游戏状态的时候也很好用。

2. 可视化的 AI 诊断

在调试中，好的工具是无价之宝，而标准调试器在诊断 AI 问题的时候总是那么力不从心。各种调试器虽然在某个具体时刻能给出很好的深度，但在解答 AI 系统怎样随着游戏进行而变化这个问题上完全无用。解决办法是在游戏里直接构造能够监控任意角色的诊断数据的可视化版本。通过将文字和 3D 线条组合起来，一些重要的 AI 系统如寻路（pathfinding）、警觉边界（awareness boundaries）、当前目标等，会较容易跟踪和查错 [Tozour02] [Laming03]。

3. 日志的作用

通常，我们在游戏里有成堆的角色彼此交互和联系，以得到非常复杂的行为。当交互失败，bug 出现之时，关键在于能够记录导致 bug 的每个角色的个别状态及事件。通过对每个角色创建单独的日志，将带有时戳的关键事件记录下来，我们就可能通过检查日志来发现错误 [Rabin00a] [Rabin02]。

4. 记录和回放的作用

像前面提到的那样，找出 bug 的关键在于可重现性。极致的可重现性需要通过记录和回放玩家的输入来实现 [Dawson01]。对于那些概率很小的死机 bug，记录和回放是找出确切原因的关键工具。但是为了支持记录和回放，你必须让游戏的行为是可预料的，也就是说对于同样的初始状态，同样的玩家输入必定会得到同样的输出结果。这并不意味着你的游戏对玩家来说是可以预知的，只是意味着你应当小心处理随机数的产生 [Lecky-Thompson00] [Freeman-Hargis03]、初始状态、输入等方面，并能在程序崩溃时将输入序列保存下来 [Dawson99]。

5. 跟踪存储分配事件

这样实现你的存储分配算子，使其对每次分配操作都进行全面的栈跟踪。通过不断地记录究竟是谁在申请内存，你将不再有内存泄漏问题需要解决。

6. 崩溃时打印出尽可能多的信息

“事后调试 (post-mortem debug)”是很重要的。程序崩溃时，理想的情况下，你会希望能够捕捉到调用堆栈、寄存器以及所有其他可能相关的状态信息。这些信息可以显示在屏幕上，写入某个文件，或自动发送至开发者的电子信箱。这一类的工具让你迅速找出崩溃的源头，只要几分钟而不是几个小时。尤其是当故障发生在美工或策划同仁的机器上，而他们并不记得是怎样触发这次崩溃的时候。

7. 对整个团队进行培训

虽然这并非一个能够编程实现的结构，但是你应该确定团队正确使用你创建的工具。请他们不要忽视错误对话框，确信他们知道怎样搜集信息从而不会丢失已找到的 bug 等等。花时间来培训测试员、美工、策划是值得的。

1.1.11 预防 bug

关于调试的讨论，若没有一段文字指导如何在第一时间避免 bug，便不能算完整。遵照这些指导方针，你或可避免编写出有 bug 的代码，或可在偶然之间发现自己不知不觉写出来的 bug。不论是什么结果，都会最后帮你排除 bug。

将编译器的警告级别 (warning level) 调到最高，并指示将警告当作错误处理 (enable warnings as errors)。首先尽可能多地排除警告，最后再用 #pragma 将剩下的警告关闭掉。有

时，自动类型转换及其他一些警告级的问题会带来潜在的 bug。

使你的游戏能在多个编译器上编译通过。如果你确保游戏用多个编译器、面向多个平台都能编译通过，不同的编译器之间在警告和错误方面的差异将保证你的代码总体上更可靠。例如，编写任天堂 GameCube™游戏机上的程序的人也可以在 Win32 下生成一个功能稍弱的版本。这也使你能够判断某个 bug 是否是具体平台所特有的。

编写你自己的内存管理器。这对于游戏机游戏是至关重要的。你必须清楚地知道正在使用哪几块内存，并对内存上溢进行保护。由于内存溢出会带来一些最难查处的 bug，首先确保不发生溢出是很重要的。在调试版本中使用预留的上溢和下溢保护内存块能使 bug 更早地暴露身份。对 PC 开发者来说，编写自己的内存管理器不是必须的，因为 VC++ 里的内存系统功能已经很强了，而且还有像 SmartHeap 之类的好工具可以用来确定内存错误。

用 assert 来检验假设。在函数的开头加上 assert 来检验关于参数的假设（例如指针非空或范围检查）。另外，如果 switch 语句的 default 情况不应该被执行到，在其中加上 assert。还有，标准 assert 可以被扩展以得到更好的调试性能 [Rabin00b]。例如，让 assert 将调用堆栈打印出来是很有用的。

总是在声明变量的时候初始化它们。如果你无法在声明某个变量时赋予它一个有意义的值，那么就给它赋一个将来一眼就能认出它有没有被初始化过的值。有时候我们会用 0xDEADBEEF、0xCDCDCDCD，或直接使用零。

总是将循环体和 if 语句体用花括号（{}）括起来。也就是将你所想的代码老老实实在地包起来，使代码所实现的功能更直观。

变量起名要容易区分彼此。例如，m_objectITime 和 m_objectJTime 看上去几乎一模一样。此类问题的典型例子是把“i”和“j”用作循环计数变量。“i”和“j”看上去很相似，你很容易把它们看混。可供选择的方法是，可以用“i”和“k”，或者干脆使用更能描述其意义的名字。更多有关变量命名的认知差异的信息可以在 [McConnell93] 中找到。

避免在多处重复同样的代码。一模一样的代码同时出现在几个不同的地方，这是不利的。如果对其中一处代码做了改动，其余几个地方不一定也会被改动。如果看上去重复代码是必要的，重新考虑一下其核心功能，尽量将大多数的代码集中到一处。

避免使用那些固定写死的“神奇数”（magic number）。当单独一个数字出现在代码中，其意义可能是完全不为人知的。如果没有写注释，就无法让人理解之所以选择这个数字的理由，以及这个数字代表什么。如果必须使用神奇数，将它们声明为有字面意义的常量或 define。

测试的时候要注意代码覆盖率。在编写完一段代码之后，应验证它的每一个分支都能正确地执行。若其中一个分支从未被执行过，那么很可能其中正潜伏着 bug。在测试不同分支的过程中你可能会发现这样一个 bug，即其中某个分支是根本不可能被执行到的。这样的 bug 越早发现就越好。

1.1.12 结论

本文向你介绍了有效率地调试游戏所需的工具。调试有时候被形容为一门艺术，但那只是由于人们越有经验就做得越好。当你把五步调试法融会贯通，又学会了识别 bug 的模式，并将自己的调试工具集成到游戏中，再形成自己在调试上的个人风格和绝招，很快地，你将

熟练且有系统地追捕到并且消灭最困难的 bug。最后再加上一点预防，我想你的游戏开发会一帆风顺，一个 bug 都没有也说不定。

1.1.13 致谢

感谢 Scott Bilas 和 Jack Matthews，他们提了极好的建议，并为本文贡献了他们的经验和智慧。人们看待调试有各自的角度，因此他们的意见在推敲本文建议的时候起了非常大的作用。

1.1.14 参考文献

[Dawson99] Dawson, Bruce, “Structured Exception Handling,” *Game Developer Magazine* (Jan 1999), pp. 52–54.

[Dawson01] Dawson, Bruce, “Game Input Recording and Playback,” *Game Programming Gems 2*, Charles River Media, 2001.

[Freeman-Hargis03] Freeman-Hargis, James, “The Statistics of Random Numbers,” *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Laming03] Laming, Brett, “The Art of Surviving a Simulation Title,” *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Lecky-Thompson00] Lecky-Thompson, Guy, “Predictable Random Numbers,” *Game Programming Gems*, Charles River Media, 2000.

[McConnell93] McConnell, Steve, *Code Complete: A Practical Handbook of Software Construction*, Microsoft Press, 1993.

[Rabin00a] Rabin, Steve, “Designing a General Robust AI Engine,” *Game Programming Gems*, Charles River Media, 2000.

[Rabin00b] Rabin, Steve, “Squeezing More Out of Assert,” *Game Programming Gems*, Charles River Media, 2000.

[Rabin02] Rabin, Steve, “Implementing a State Machine Language,” *AI Game Programming Wisdom*, Charles River Media, 2000.

[Spolsky02] Spolsky, Joel, “The Law of Leaky Abstractions,” *Joel on Software*, 2002, available online at www.joelonsoftware.com/articles/LeakyAbstractions.html.

[Tozour02] Tozour, Paul, “Building an AI Diagnostic Toolset,” *AI Game Programming Wisdom*, Charles River Media, 2002.



1.2 一个基于 HTML 的日志和调试系统

作者: James Boer

E-mail: james.boer@gte.net

译者: 沙鹰

审校: 万太平

由于程序输入条件的无规律和实时的本质, 调试游戏程序比调试传统的应用程序更有挑战性。在许多应用程序中, 精确重现使 bug 发生的步骤是轻而易举的。可是, 游戏, 通常有一些额外的不利因素, 例如不可预知的用户输入、复杂的实时游戏环境, 还有零星的不确定性因素, 如人工智能等。常常, 等到 bug 被发觉时, 错误事件所指示的对出错负有责任的数据或变量的值早已改变。而且, bug 通常是由测试员在运行一个没有调试器的发布版本 (release build) 上发现的。那么, 开发者怎样才能抓到这些难以捉摸的 bug 呢? 我们可通过保存事件日志得到潜在的答案。

1.2.1 于日志系统的优势

我们都曾从负责测试的人那儿听说过这一类的 bug: “在基诺船长 (Kaptain Keeno) 那关, 也就是第 6 关, 快要结束的时候, 我遇见了一大群 Zarbovian 喷火兵, 我在做出超级跳动作的同时用 KaBlaminator 射线枪干掉了其中最后的那个机器人, 可是之后全部机器人竟然开始大跳特跳 Macarena 舞。”诸如此类的 bug 是最难找到并修改的 bug。其重现步骤即使不是难度超高, 就是过程极端冗长。就算成功地重现了一个 bug, 你可能只是陷入这样的疑惑——是谁把代码写得这么奇怪, 叫人怎么读呀?

而这正是日志系统所擅长的。只要将关键事件、变量和统计这些连续的报告输出为具有格式的文本文件, 程序员就能够了解在代码中大概发生了什么而导致 bug 发生。日志系统给了我们随着时间的过去不断观察数据的能力, 且无需手动单步执行大量的代码。

1.2.2 究竟什么是事件日志?

提到“事件日志 (event logging)”这个说法, 我们指的是捕捉实时事件和消息以便随后获取和分析的任务。在最简单的情况下, 日志可能只是通过基本磁盘函数实时地保存游戏中的信息, 例如变量内容和消息描述等。例如, 这使开发者能够了解变量内容随时间变化的历史过程, 而不只是通

过调试器获取的该变量在单一时刻的内容。说实话，尽管事件是实时地被捕捉和记录到日志，日志系统仍然可以归类于“事后调试（post-mortem debug）”，这是因为该系统擅长于记录已经发生过的事件。可是，与大多数设计用来捕捉应用程序崩溃的直接原因的“事后调试器”不同，日志系统亦擅长捕捉行为 bug，条件是有足够多的和代码有关的数据被记录下来。

那么，如果你暂时还没有在一段混乱的、需要调试的代码中进行记录数据呢？尽管去添加你所需要的日志信息吧，然后耐心等待问题重现。事件日志并不是万灵药——它只是你调试工具箱里的另一把扳手罢了，始终是需要程序员积极计划和参与的。不过，日志文件的妙处就在于，通常它和其他模块完全不会冲突，因此随时随地使用大量的活动日志记录也没有问题。当你一旦为系统增加了信息日志消息，通常就不再有什么借口把消息去除。这样一来，此系统用得越多，产生的日志也就会包含越来越多的关于游戏状态的信息。

日志系统应如何工作？没有一种设计是十全十美的，在此我们介绍一种对大量杂乱的调试数据进行高效率排序的方法。

1.2.3 HTML 和调用堆栈

坦白地讲，只是要实现一个尽职尽责地将数据记录到一个文本文件中的日志系统并不难。但不幸的是，这样导出的原始数据（raw data dump），由于绝对数量过大，将是难于解读的。因此我们需要一个方法，将某些消息以一种能起到提示作用的格式，与其他消息区分开来。同时，各事件的相对时间也应当被保存，以便了解某事件在记录下来的消息序列这个参照系中的位置。如此，日志系统便能为记录下来的事件提供上下文，从而简化在游戏源代码级别的跟踪逻辑。开发者们一般都在其调试器中显示调用堆栈，并以此作为判断某段代码被运行到的首要方法。在我们的日志系统里，也会利用一个调用堆栈来识别每条消息的位置和发源地。最后一点要注意，和其他所有调试工具一样，在最终的（或任何中间版本）可执行文件中，日志系统应当能够不费力气地完全编译。

除了这些技术要求以外，还须注意日志系统应当易于集成、易于使用。经验告诉我们，一个功能强大但是复杂的系统，尽管从软件工程的意义上讲可能是个奇迹，但是很可能得不到使用。“简单和实用”是调试工具的关键。

我们将尝试用三种主要机制解决数据分析问题。第一种，也是最显而易见的解决方法，是对数据采取 HTML（HyperText Markup Language，超文本置标语言）的格式。采用 HTML 有几个优点：它是已制定的基于文本的标准，基本的格式规则很简单，也可以灵活地支持几乎所有我们需要的格式类型，而且现在所有的计算机系统都安装有 HTML 阅读器，或者叫 Web 浏览器。通过使用 HTML，我们可以相对简单地完成各种格式化任务。其中最重要的，可能就是我们能够容易地将记录下来的事件的调用堆栈以一种缩进格式表现，请看程序清单 1.2.1。该清单显示了某个棒球游戏生成的日志输出范例。

程序清单 1.2.1 日志输出范例

```
HittingBrain::updateSwing()  
    HittingBrain::updatePitchData()
```

```

HittingBrain::calculateSwingTimingAndPosition()
Expected pitch time: 0.857448
Expected pitch location: (-8.150428, -8.516977)
Actual pitch time: 1.244251
Actual pitch location: (8.695169, 7.850054)
Update = 38185, Game Time = 00:10:36.41

```

由于有了一定的上下文，你就能看出，在游戏中，此时此刻（约在游戏开始后 10 分半钟）发生了什么。你看到一个类的成员函数 `HittingBrain::updateSwing()` 调用了 `updatePitchData()` 函数，后者调用了 `calculateSwingTimingAndPosition()` 函数。这正是提取出日志信息的地方：AI 正在尝试击球。你可以猜测 AI 在期待对方投出一个内角高目快速球，而实际上对方投出的是外角慢速变化球。从此数据可见，AI 击球手多半会因过早地挥棒而把球漏掉。利用这样详细的历史数据，我们终于能够看到游戏内部的过去，检视游戏中实际发生的事件，然后了解问题是什么。

另外，尽管没有在程序清单 1.2.1 中表现出来，但每条日志信息的文字都可以是彩色的。一个合理的着色方法是定义类别，让每个类别具有自己独特的颜色和式样，这样一来你就比较容易在日志文件里的众多消息中找到你所关心的那些。

1.2.4 工作原理



虽然在本书的配套光盘里有一套完整的日志系统源代码，其中有些有趣的模块值得在这里详细介绍一下。事件记录器的使用很简单，基本上分三步，常规作业（housekeeping）、函数跟踪和消息记录。我们先看一下常规作业，其中包括初始化、更新和关闭系统。程序清单 1.2.2 给出了其中部分源代码。

程序清单 1.2.2 初始化和关闭日志系统

```

// 在程序初始处调用 LOG_INIT
LOG_INIT("gamelog.html");

//...

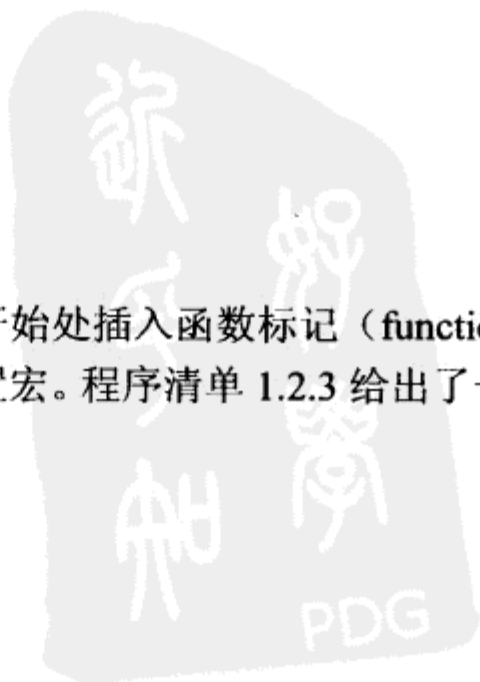
// 在每个更新循环中调用一次 LOG-UPDATE
// m_GameTickDelta 是一个代表自上一次更新 (tick) 以来
// 经过的时间长短（以秒为单位）的浮点型变量
LOG_UPDATE(m_GameTickDelta);

//...

// 在程序终止处调用 LOG_TERM
LOG_TERM();

```

接下来，你要在每个需要跟踪的函数体的最开始处插入函数标记（function marker）。通常这表示要在游戏的大多数主循环更新函数里放置宏。程序清单 1.2.3 给出了一例，包括一条简单的日志信息。



程序清单 1.2.3 放置日志宏

```
SomeClass::DoSomething(int value)
{
    FN("SomeClass::DoSomething()");
    LOG("A value of %d was passed to this function",
        value);
}
```

尽管要在每个函数里调用一次 FN 宏看上去有些冗长，但这点额外的工作将对有关日志消息源自何处提供很大的帮助。在我们所见的本函数中添加宏，再在调用此函数的函数里添加宏，我们将会获得一系列看上去和程序清单 1.2.1 中的结果相似的日志消息——你可以简单地在这棵函数名称树上回溯，找出函数之间的调用关系。

你会注意到在这里我们使用宏来掩藏了一些实现细节，这让代码输入变得简单，也更容易读懂，还可以让我们在生成最终版本的时候很容易拿掉整个日志系统，就像标准的 assert 宏一样。

跟踪调用堆栈的系统利用一个小的临时对象来纪录函数域。程序清单 1.2.4 揭示了 FN 宏是如何在每次被用到时创建一个小对象的。

程序清单 1.2.4 FN 宏的实现

```
#define FN(var_1) EventLogFN obj__scope(var_1)

// 用于压栈和退栈的辅助函数
class EventLogFN
{
public:
    EventLogFN(const char* szFunction);
    ~EventLogFN();
};

EventLogFN::EventLogFN(const char* szFunction)
{
    g_Log.pushFunction(szFunction);
}

EventLogFN::~~EventLogFN()
{
    g_Log.popFunction();
}
```

因为这个对象是如此的轻量级，自身没有数据，又是在栈上被分配的，我们可以确信这些宏将只带来最轻程度的负担。为了更好地保证操作的轻便性，我们限制了字符串只能是一个常量字符指针。通过在此处代码中使用常量字符指针，日志系统避免进行内存分配和字符串拷贝操作，将系统开销削减至仅有往栈里压入和弹出单个指针那么一点点。也因为传入的是常量字符串，我们可以不需要特别优化。因此，千万记住不要传入临时字符串数据，例如标准字符串对象的 c_str() 方法的返回值。

因为我们将日志条目输出为 HTML 格式，如果没有利用 HTML 标准的多种色彩和格式，那将是很羞人的。着色后的输出可以区分不同类型的消息，还可以按其所属逻辑代码模块分别列出，因此非常有用。程序清单 1.2.5 给出了我们的日志系统的格式标志集，这些格式可被应用到任意消息。

程序清单 1.2.5 日志的格式标志

```
#define LOG_COLOR_RED      0x00000001
#define LOG_COLOR_DK_RED   0x00000002
#define LOG_COLOR_GREEN    0x00000004
#define LOG_COLOR_DK_GREEN 0x00000008
#define LOG_COLOR_BLUE     0x00000010
#define LOG_COLOR_DK_BLUE  0x00000020
#define LOG_BOLD            0x00000040
#define LOG_ITALICS         0x00000080
#define LOG_UNDERLINE       0x00000100
#define LOG_PRINTF          0x00000200
#define LOG_DEBUG_OUT       0x00000400
#define LOG_DISABLE         0x00000800
```

有些格式改变消息文本的颜色或外观，还有一些将文本重定向到不同的调试输出区，例如 stdout 或某个调试窗口。通过将这些标识中的一部分 OR 在一起形成一些单个的#define，你可以很容易地将消息分类，不同类别的消息在输出流中具有各自独特的外观。

类别#define 可以作为调用重载的 LOG 函数时的第一个参数。你一定注意到了我们选用一个简单的 printf 风格的系统来处理数据格式化和多重参数。程序清单 1.2.6 示范了使用预定义的格式标志后，典型的输出日志消息的代码是什么样子。

程序清单 1.2.6 日志调用示例

```
#define LOG_AI      LOG_DEBUG_OUT|LOG_BOLD|LOG_COLOR_RED
#define LOG_AUDIO LOG_ITALICS|LOG_COLOR_BLUE

LOG(LOG_AI, "Test AI message");
LOG(LOG_AUDIO, "Test Audio message");
```

这两处日志输出的结果是两条具有不同的颜色和格式的消息，这一来就很容易区分彼此。

实际的 HTML 输出是很容易生成的。HTML 中的格式命令用“小于号”和“大于号”分隔开来，像这样：<command>。一般来说，能够缩写的命令都被缩短成一个字母。例如，开始一个新的段落的 HTML 命令是<p>。在 HTML 里，每一个命令都包含一对格式命令。作为结尾的格式命令和作为开头的格式命令是严格相同的，只不过在命令标识符前面有一个斜杠，像这样：</p>。我们只用到了一些很基本的命令，包括段落、字体、不排序的列表，当然还有基本的文本格式修饰符，如**粗体**，*斜体*，和下划线。

这些命令中的每一个（也包括其相应的结尾命令）都被封装成为日志类的一个成员函数。这样就很容易将一系列 HTML 格式命令排成一列，如程序清单 1.2.7 所示。例子中的代码来自 EventLogger::LogOutput()函数，示范了如何使用传入标准日志消息的格式标志，输出一行经过格式化的文本。

程序清单 1.2.7 用多个格式调用组成一行日志信息

```
// 将经过格式化的 HTML 数据写入输出缓冲区
writeIndent();
writeStartListItem();
writeStartFont("Arial", 2, r, g, b);
writeText(nFlags, m_szLogBuffer);
writeEndFont();
writeEndListItem();
writeEndLine();
```



为了防止输出缓冲被无用的数据填满，我们做了一条重要的优化，即仅显示与实际被输出的消息有关的数据。换句话说，日志系统并非每帧都进行输出，而是只在日志消息被记录的那一帧里才输出。同理，我们也并不跟踪和输出完整的一帧的代码路径。限制堆栈只打印那些为了显示消息被记录的位置所必需的路径。这两条优化帮助从最终的日志文件中剔除无用的信息，也使最终结果有较好的可读性。你可细看随书光盘里的代码具体是怎么实现此功能的，主要也就是在函数堆栈里记录位置。

1.2.5 一些有用的心得

像对任何其他工具一样，理解本系统的长处和弱点是很重要的，这样才能最有效率地使用它。下面是一些心得和建议，供你参考。

- 本系统并非线程安全的（thread-safe）。为了要实现线程安全，堆栈调用要在每个线程上进行，或利用一个独立的对象跟踪来自各线程的数据。
- 在全局对象的构造或析构函数中添加日志消息或函数标记时要小心。因为多半这些函数会在日志系统被初始化之前就被调用，从而导致日志程序失效。
- 对每一帧都跟踪变量并不是一个理想的用法（尽管日志系统能够这样做）。日志系统更适用于跟踪间歇的事件，例如记录一个 AI 实体的状态变化。每帧都输出变量会带来问题，游戏长时间运行后日志文件会变得巨大。
- （在此实现的）系统并不特别适合捕捉死机 bug，相比较之下，它对捕捉行为异常的 bug 更有帮助。
- 和使用任何调试工具一样，程序员有责任巧妙地使用本系统。放置所有函数标识宏以及巧妙地将相关数据写进日志，这需要相当的时间投入。
- 除了显然可以用作调试工具之外，此日志系统还能极大地帮助程序员理解项目中一大段代码的基本执行流程。

1.2.6 结论

无疑，事件日志并非实时调试技术的最新成就，但它可以是你的调试锦囊中的一个重要工具。它的价值体现在协助捕捉到在问题发作之前发生在程序中的事件——这是使用传统调试器所不容易做到的。而且，由于我们使用了标准文件格式，就算数据输出变得再有创造性，

阅读器将依然能够显示数据。例如，你或许想使用层叠样式表（CSS，Cascading Style Sheet）来定义不同类型消息的属性，使其他程序员可以高亮显示输出中的一些部分，或甚至转而使用 XML 以更有效地操纵大量数据输出。甚至你可以避免使用浏览器作为阅读器，例如以 Excel 之类的电子数据表软件能够接受的 tab-或逗号分隔文件的格式输出。

每个项目都有其不同的需求，你应当仔细思考，如何才能最好地捕捉那些能用来指示错误发生于何时的运行时数据（runtime data）。像本文所描述的这样一个日志系统，细心地使用，将是你发现和解决一些最难的 bug 的有效捷径。



1.3 时钟：游戏的脉搏尽在掌握

作者：Noel Llopis, Day 1 Studios

E-mail: llopis@convexhull.com

译者：沙鹰

审校：万太平

这游戏里，一切都是由时钟（clock）来驱动的。时钟使时间向前，而正是这流逝的时间让游戏世界中的万物运动起来，使镜头跟随着玩家，甚至在游戏结束后使开发者名单不断向上卷。不幸的是，“每当游戏需要时就直接查询某些系统定时器来取得当前时间”这样天真的做法伴有很多问题。下面是几个问题的例子。

- 在一帧内的不同地方查询时间会返回不同的值（导致屏幕上的对象在更新时产生奇怪的扭曲失真）。
- 暂停游戏，但是允许游戏的一些部件继续运行会有问题的。例如，当游戏暂停时，用户界面动画和旋转中的镜头仍然处于动态。
- 帧速率的变化会导致动画以及镜头运动有容易察觉的抖动。

1.3.1 关于时间的基础

在很久以前，那是游戏开发的黑暗（无知）年代，大多数程序员根本不使用时钟。他们将尽可能多的东西放在一帧内执行（通常帧的长度由 CPU 能够执行多少计算来决定，或由相邻的两个显示器垂直同步信号（vertical sync signal）之间的时间决定）。这样做的效果不错——只要游戏总是在同等配置的电脑上运行就没问题。但如果有人试图在较快的 CPU 上运行此游戏，整个游戏都会变快。而且不单是游戏的显示帧速率加快而已，实际上一切物体在屏幕上的运动都加快了，以至于游戏变得没法让人玩下去。

在今日，过去的早已烟消云散。大多数现代的游戏都利用某种形式的时钟来驱动游戏，并使游戏运行的速度独立于运行游戏的系统的速度。即使是运行平台硬件配置相对固定的电视游戏机游戏，也能从使用时钟系统中获益，例如根据 PAL（Phase Alternating Line）或 NTSC（National Television System Committee）视频制式以不同的频率来刷新游戏。

大部分游戏都运用主游戏循环（main game loop）的机制。只要游戏还在运行，这循环就会反复执行，每帧一次。一帧内的所有处理都必须在循环体内进行。下面列出一些一般在帧内处理的主要事件。

- 处理用户的任意输入。
- 运行人工智能 (AI)。
- 接受网络分组数据包 (Packet)。
- 更新游戏世界中的所有对象。
- 渲染所有当前可见的对象。

以上各步骤可依不同先后顺序执行。有一些游戏的结构较为复杂,例如每隔几帧才执行一次游戏的 AI 或模拟部分,以便使图形渲染部分与游戏的其余模块之间的耦合度降到最低。还有一些游戏具备多通更新 (multiple update pass), 或一个特别的处理消息的步骤。没关系,说到底,它们还是与我们描述的基本主循环非常相似。

典型来说,在循环体内进行计算的类型包括:判断从某对象上一次被更新以来已过了多久时间;计算从上次更新的时刻起有哪些变动是必须更新的;为反映正确的状态而进行更新。由于这些计算通常在每一帧中进行,所以可以将上一帧的长度近似地看作对象上一次更新以来的时间。

一般来说游戏有两种处理事件的方法可供选择,可变帧长度或固定帧长度。绝大多数 PC 游戏,和许多 console (家用电视游戏机) 游戏,均采用可变帧长度的方法。也就是说,每帧的时间可以各不相同,具体长度常因当前屏幕上显示的内容变化而变化。

采用固定帧长度方法的游戏多见于游戏机平台,这是因为硬件配置已知,可以假定每帧总是都具有相同的长度 (通常与显示的垂直同步信号成倍数关系)。即使你的游戏采用固定帧长度方法,并因此而缓解了许多我们下面即将讨论的问题,了解本文介绍的明白易懂的时钟组成及一些功能还是有益无害的。

1.3.2 时钟系统的组成

我们要开发一个具备如下功能的时钟系统。

- 可靠地获取当前时间、一帧的长度。
- 使游戏时间值的暂停与缩放独立于程序的其他部分。
- 时间归零,或用任意值重设时钟。
- 支持可变帧长度,也支持固定帧长度。
- 避免因连续几帧的长度彼此差别过大而产生失真 (artifact)。
- 避免精度问题。

不妨从两个不同的概念入手,时钟和定时器 (timer)。

游戏中只有一个时钟,由它驱动所有的定时器。时钟取得当前的时间 (单调递增的)。不可以暂停或直接操作时钟,就和我们不能控制真实的时间一样。

不像时钟只有一个,定时器可以有很多个,为各自在游戏中不同的用途而被创建。定时器由时钟驱动,但可以被用户进行操作:可以暂停、归零 (reset)、以比例加快或放慢。游戏可以创建一个定时器来记录世界时间,创建另一个给 GUI 用,再来一个用于电影回放,如此种种不一而足。显然,每个定时器对当前时间,甚至可能对帧的长度,有其独特的诠释。

时钟究竟从哪里获得时间的值呢?答案是通常从某种高精度的系统定时器处取得时间,具体实现方法通常是与平台相关的。总之,可以把取出时间值的操作抽象为叫做“时间源 (time source)”的逻辑概念。时间源可以是使用与平台相关的定时器、读取文件、甚至根据垂直同步

信号来计算时间。无论采用那种方法，我们的时钟都以相同的方式工作，独立于特定的时间源。
很方便，上述的三个概念：时钟、定时器、时间源，可以各自用一个 C++ 类来表示。图 1.3.1 是刻画三个类之间关系的 UML 图。

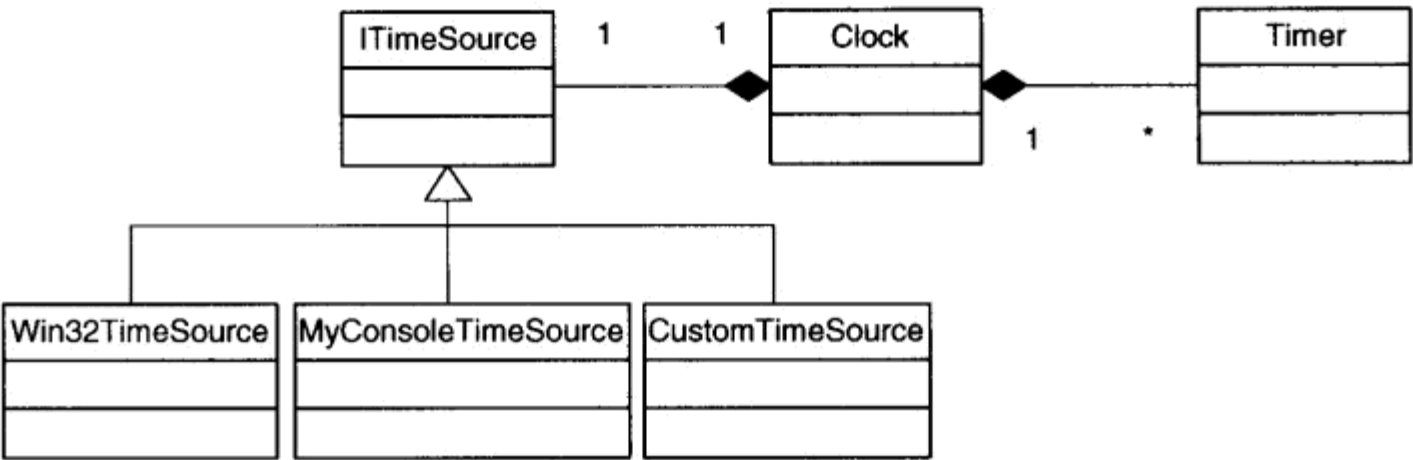


图 1.3.1 时钟系统中的类的 UML 类图



在本书的光盘上可以找到这些类的完整实现，以及本文介绍的一些其他功能，外加一个单元测试集。

在每帧开始的时候，调用时钟类的成员函数 `FrameStep()`。它使时钟从时间源获取更新的时间读数，更新内部时间值，并更新所有与该时钟有依赖关系的定时器。当该函数被调用后，时钟里的当前时间以及上一帧的长度将保持不变，直到下一帧开始时再次调用该函数时才被更新。这不但提高了效率——因为每次查询定时器时无需做额外的计算，而且允许我们以一个稳定可靠的参考帧来更新这一帧中的所有对象。这解决了第一个问题：在一帧内的不同地方查询时间保证返回相同的值。

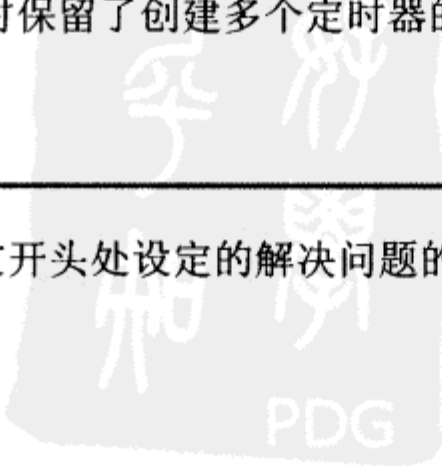
因为定时器彼此互相独立，我们可以自由地操作其中任意的定时器，而不必担心会影响核心的时钟或其他定时器。我们可以暂停某一个游戏定时器，从而使游戏中所有的动作都停下来，但是与此同时，所有用户界面的控件仍将正常地以动态表示，因为它们使用的是另一个 GUI 定时器。我们甚至可以尝试将某个定时器加快或放慢，以制造某种惊人的视觉效果，例如以慢动作回放的爆炸、主观视角或第一人称视角（subjective viewport），或在动作几乎停滞的情况下让摄像机（camera）高速运动。

该系统提供的另一个功能是，允许对时钟系统设置我们自己的值。这很容易实现，使用一个包含我们所需要的新的时间源就可以。但为何要这样做呢？比如说，我们可能需要将一些为分镜头（cut scene）记录下来的精确时钟数据输入时钟系统，以便实现精确的回放（假定原始的时钟值也是分镜剪接数据的一部分）。

总之，若游戏采用固定帧长度的方案，可以实现一个非常简单的、总是返回我们期望的帧长度的时间源，同时保留了创建多个定时器的功能。

1.3.3 避免失真

至此，围绕在本文开头处设定的解决问题的目标，我们已经提出了许多有关时钟的问题。



但还有一个麻烦的问题没有提到，就是时间带来的失真问题。该问题主要原因有三，下面让我们分别讨论。

1. 峰值

用一点时间思考我们究竟是要测量什么？明确地说，获取帧长度（frame duration，每一帧所用的时间）究竟意味着什么？我们真的要知道从上一帧的开始（即上一次更新对象时）到现在已经精确地流逝了多少时间吗？

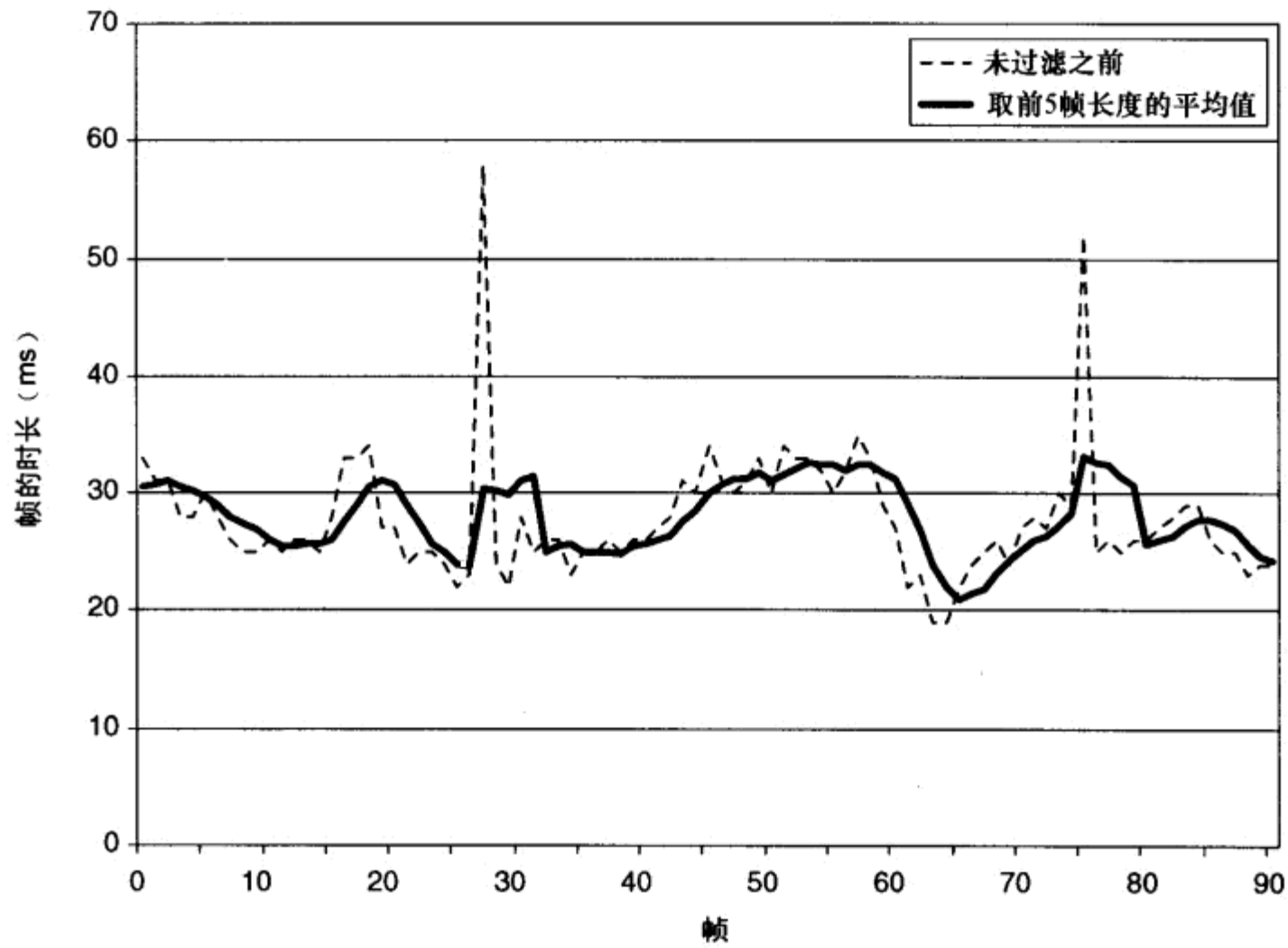
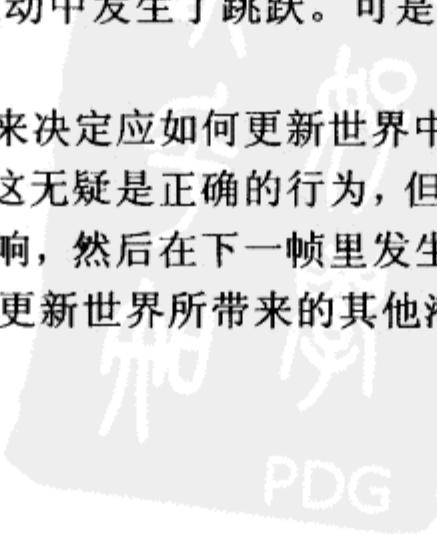


图 1.3.2 从某游戏的某部分中采样的每帧长度图。虚线表示未过滤的帧长度，实线表示最近 5 帧长度的平均值

图 1.3.2 显示了一系列连续帧的时长图。其中特别需要注意的是，总的说来是平稳的虚线上却有两个峰值。这样的峰值一点也不稀奇，尤其是在 PC 游戏中，因为可能会频繁地进行硬盘读写，或显示卡正在载入大量纹理贴图（texture）。

举例来说，有个物体，比方说是一辆汽车，正在以常速行驶穿过屏幕的时候遇到了帧长度的峰值，那么会怎么样呢？若是某一帧用了特别久的时间，我们对此是无能为力的，而且会因此察觉物体的运动中发生了跳跃。可是，在下一帧里，汽车应当以常速前进才对。

若我们采用上一帧的长度来决定应如何更新世界中的对象，这辆车会突然向前跳跃以便补偿在峰值中损失的时间。这无疑是正确的行为，但它确实是我们所希望看到的吗？首先，用户受到一个长的帧的影响，然后在下一帧里发生了跳跃。这看起来一定好不了，更不用说用一个大的时间间隔来更新世界所带来的其他潜在问题了（物理和碰撞问题，或模



拟的反馈回路)。

大多数时候,若是能够防止时间的变化量(Δt)过于剧烈,我们就可以获得不错的结果。一个简便的实现方法就是取之前 n 帧长度的平均值,而不是仅仅依赖上一帧的长度。让 n 取 5~10 之间的值较为理想,因为这样的值对于使平均帧长度较好地响应帧速率的变化来说足够小,对于避免高的峰值来说又足够大了。图 1.3.2 中的实线代表了对实际帧长度(虚线)取前 5 帧进行平均的结果。

若是出现了巨大的峰值:例如长达 10 秒或 5 分钟的峰值呢?就算取了前 10 帧进行平均,结果仍将巨大。

为什么会出现如此巨大的峰值?最常见的场合就是当你通过调试器在游戏循环中设置断点并中断下来的时候。你先观察了几个变量的值,又用单步(step into)调试了几步,在 5 分钟过后你终于决定继续执行游戏。可是此时除非你已经采取了一些对策,否则下次得到的帧长一定大得吓死人。很有可能,一切对象都突然移动了很远,并且速度是如此地快,几乎是直奔场景外面而去。而且除非你对物理模拟计算采用固定的步幅,否则所有的计算都报废了。

限制帧长度的上限是一个简单有效的对策。若我们的游戏应该运行在每秒钟 30 帧,也就是每帧 33ms,那么我们(在通常意义上)就绝不应该有一个长于 250ms 的帧。那就等于 4fps! 因此,通过设置上限为 250ms,在调试器里中断然后继续的下一各时间间隔就会是 250ms(通过与前几帧进行平均,结果会小很多),这样就能够正常地继续运行游戏了。

2. 垂直同步

时常,我们希望等到下一个垂直同步(vertical sync)发生才将下一帧显示出来。这样做可以防止由于显示器或电视机的刷新频率与游戏显示刷新频率不同而产生的锯齿现象(tearing)。在等待中,通常的操作模式是进行一切在帧内进行的处理,然后阻塞(block),等待下一个同步。当它到来时,将后备缓冲(back buffer)中的内容显示出来,开始下一帧。

这个想法是好的,但它与现代 PC 和游戏机中高度并行的图形系统发生冲突。越来越常见的是,所有发送到图形处理器中的命令进入队列等待着尽快被硬件处理,立即返回游戏继续执行。这意味着,一旦不小心,就可能陷入非常恼人的境地——帧长度的变化范围相当大。此时,简单的进行平均并不能给我们足够的帮助。

考虑这样的情形:我们在一个 60Hz 的显示设备上等待每一个垂直同步信号,希望能够维持 30fps 的稳定速度。但是,由于玩家视线的方向上是游戏世界的一个非常简单的部分,渲染和更新只需要很少的一点时间。也就是说我们将在几个毫秒的时间内处理完毕所有的更新,然后通知图形处理器在下一次垂直同步发生的时候交换缓冲。由于该命令进入了一个队列,执行随即返回到游戏中,也就是说随即可以开始下一帧了。

问题来了。由于上一帧的实际长度只有几毫秒,这一短短的时间是我们对这帧更新对象状态的依据,然而在视觉上,我们是以 30fps 的速度显示场景。在几帧过后,图形处理器的命令队列可能已经满了,因而在队列基本清空之前会导致某帧的长度变得特别长。

这个问题表现为跳帧，动画不平滑。虽然帧速率固定在 30fps，世界中对象的运动仍然有着容易察觉的不连续。

如果在你的平台上出现了此问题，较好的解决方案是在交换后备缓冲的代码前后增加某些与平台相关的代码段，以避免游戏超前图形处理器一帧以上。这样一来，时间间隔会变得比较规律，同时仍然能够利用硬件的并行性。

3. 精度

处理时钟和定时器的另一个棘手的问题就是精度。应当如何表示时间，又会有什么样的误差呢？

即使你所选用的平台具备超高分辨率的系统时钟，能够返回从古生代开始的时间，而且单位是纳秒，我们仍然需要解决游戏里不连续的时间值的表示问题。我们要用一个浮点数来表示从本关开始起的以秒为单位的时间呢？还是用一个长整型来表示毫秒数呢？或是采用其他截然不同的方法呢？答案是，这取决于你的游戏和你的平台，因此建议事先了解和权衡不同的可能性。

将逝去的时间以毫秒为单位保存在一个整数中，这对那些曾涉猎系统编程的朋友一定是熟悉的。这正是为人所熟知的“嘀哒 (tick)”界面，一个 tick 就代表一毫秒。该方法有稳定的精度，但同时也是相当粗略的。此方法可表示的最小时间单位是 1ms。考虑到 60Hz 刷新频率下的一帧的时间是 16.6667ms，采用此方法几乎相当于忽略了半个毫秒之多。另一个主要不足是，采用 32 位长整型也只能表示 49 天的时间。若我们的游戏预计要持续运行更长的时间，则需要采用不同的方案，或利用包裹类 (Wrap-around) 机制。此种表示法的最后一个缺点是，有时候，在进行物理计算甚至是简单的位置更新的时候，用整数来表示毫秒数处理起来并不是很方便。很有可能需要先行转换成浮点数才能对时间进行处理。

采用浮点数来表示单位为秒的时间，在大多数计算中是更加自然的。不过，浮点数也有其固有的精度问题。与整数不同，浮点数在表示较小的值的时候具有较高的精度，当值逐渐加大，精度也缓缓降低。采用 32 位浮点数的话，仅仅 4 小时之后，对时间的分辨率就降低到 1ms 左右，而 3 天后分辨率将降低到 15ms，勉强能够表示 60Hz 刷新率下的一帧长度。显然的，3 天后情况就越来越糟了。

双精度浮点数，也就是 64 位长的浮点数，是较可信赖的表示法。位长足够长，可以坚持很久而不至于丧失精度到不可接受的程度；作为浮点数又便于数学运算。缺点是在 32 位的系统上进行 64 位浮点运算的代价较大。

在 64 位平台投入实用之前，较好的策略可能是在内部保持 64 位的精度，但在游戏中支持用各种表示法获取时间值。特别要提到的是 float 类型很适合用来表示帧长度，这是因为帧长度的值通常较小，在值的范围内浮点数的精度足以胜任的缘故。

图 1.3.3 表示了分别用 32 位整数、浮点数、双精度浮点数表示同样时间间隔的精确度。请注意 y 轴 (精度) 采用对数刻度。采用整形将在整个范围内维持 1ms 的常数精度；浮点数“迅速的”丢失精度，但在 16 个小时之内都保持在 5ms 以内。而且，在最初的 4~5 小时内，浮点数的精度比整数更好。双精度浮点数的精度最高，这也是为什么该图的精度轴必须采用对数刻度的原因：即使在 80 小时过后，它的精度仍然高达 0.00000023ms！

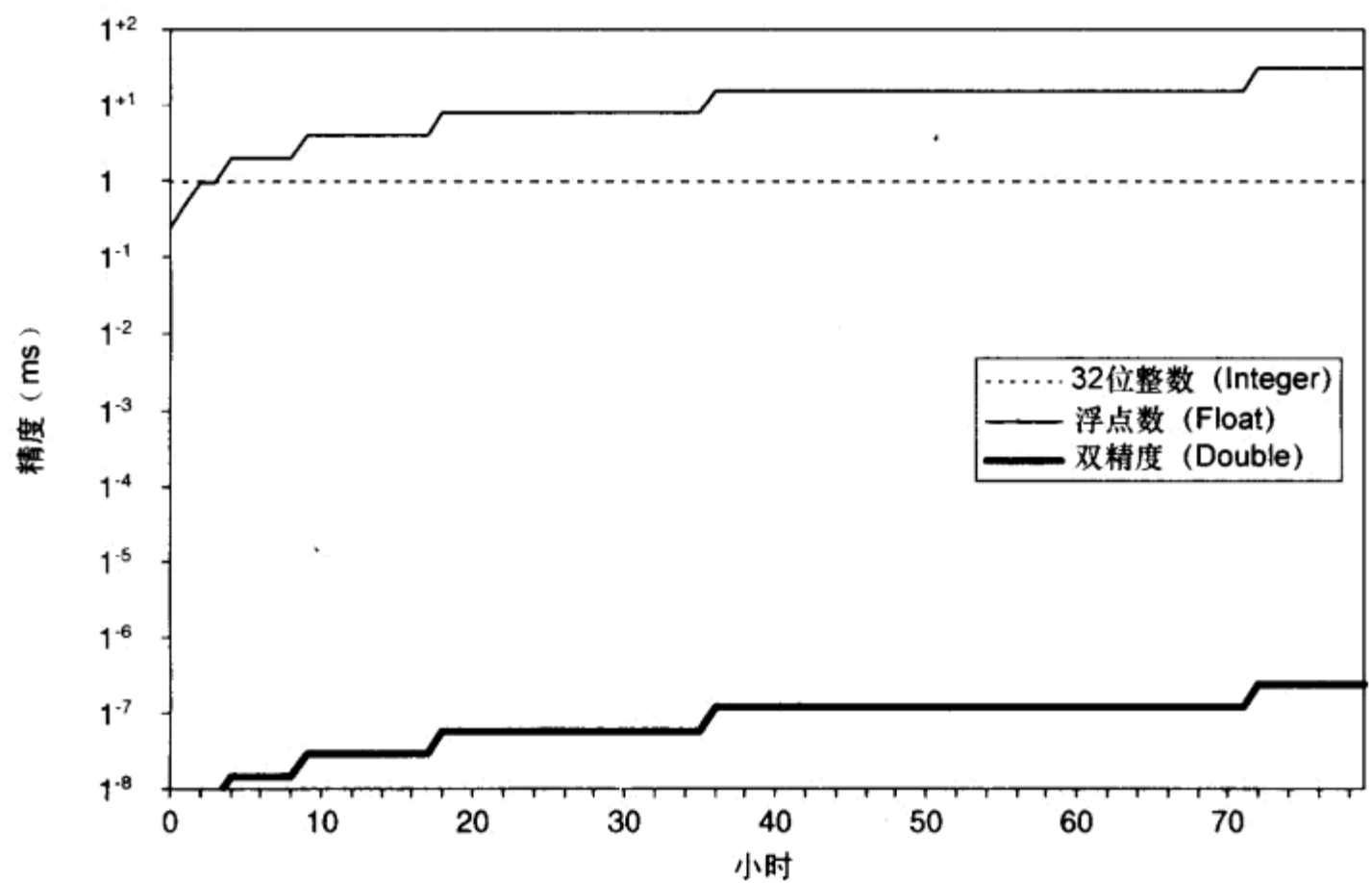


图 1.3.3 三种不同的时间表示法的精度

1.3.4 结论



处理游戏中时钟的技术，本文中涉及的只是一部分。本文清楚地描述了一个时钟/定时器系统，并解释了主要的问题以及如何防治。在随书光盘上提供的源程序是个可用的实现，随之还有一个完整的单元测试集。

对此系统可以做一些有意思的扩展，例如使某些定时器依赖于另一些定时器，创建一棵层级结构的依赖关系树，从而通过对公共根节点上的参数进行修改，达到同时操作整个定时器组的目的。若是你的游戏里包含很多时间尺度的变化，例如电影《黑客帝国》里那样的子弹飞行轨迹或爆炸的慢镜头特写，这将是尤其有用的。



1.4 设计和维护大型跨平台库

作者: David Etherton, Rockstar San Diego

E-mail: etherton@rockstarsandiego.com

译者: 沙鹰

审校: 万太平

在当今的游戏市场上, 项目预算越来越高, 因此发行商通常指望在多个游戏机平台上发行同一款游戏的多个版本, 来获得较大的投资回报。尽管可以将移植项目外包给其他一些独立的开发工作室, 但这样的话要冒最终产品质量下降的风险。如果从一开始就考虑到移植工作, 则会比较容易开发出质量较高的游戏。可是出版商通常不乐意为游戏的多平台移植版支付过多额外的支出, 因此对游戏开发者来说, 最好是尽可能地并行进行移植开发。

1.4.1 设计

在动手开始写代码之前, 你需要规划库的基本结构。一个大工作室的中型规模的项目, 可能包含大大小小几十个库, 分布在共享引擎及与项目直接相关的代码中。其中一些子系统比其他的更为复杂, 因此应当尽可能地将功能划分成数个彼此相关的模块。例如在物理库中, 将碰撞检测 (collision detection) 和碰撞响应 (collision response) 分在不同的模块中。

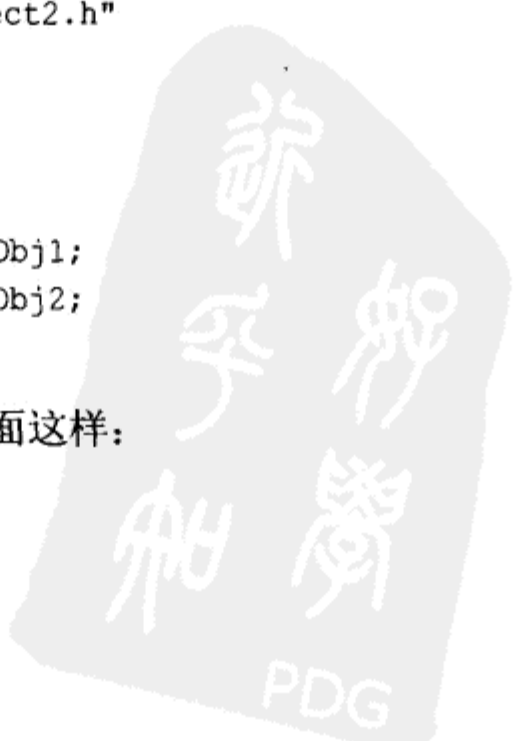
尽可能地将接口与实现分离。同时避免让你的接口直接依赖于其他的接口。

```
// Class.h
#include "object1.h"
#include "object2.h"

class Object3
{
private:
    Object1 m_Obj1;
    Object2 m_Obj2;
};
```

最好改写成下面这样:

```
// Class.h
class Object1;
```



```
class Object2;

class Object3
{
private:
    Object1 *m_Obj1;
    Object2 *m_Obj2;
};
```

有时候出于效率的需要，不适合采用后面一种写法。例如当你需要直接包含子对象以便将更多的运算写成内联函数的时候。

如果你用 C++ 编程，就不应当声明公有成员变量，而应编写用来取得成员变量值的函数。这使你得以在不影响接口的情况下对实现进行修改。可能的话，考虑使用类工厂（Class Factory）模式。这种做法的主要缺点是几乎每一个入口点都是一个虚函数，这在一定程度上影响了性能，特别是在那些数据缓冲尺寸较小的平台上。

当你的代码逐渐变得复杂和庞大，你应当果断地去除那些没有被用到的类，并消除那些总是一再出现的循环依赖关系。例如，如果有两个类彼此需要对方的数据，就考虑将它们所共享的数据移到一个新的单独的类中，并让这两个类直接存取这个新类。如果层次较低的代码需要调用层次较高的代码，试试看用函数指针或虚函数工厂来消除依赖关系。详细叙述可以参见 [Lakos96]。

为了避免符号名的重复，也为了文档可以有一个基本层次，请将你的模块划分为用简短的前缀命名的名字空间：gfx 代表图形（graphics）、ph 代表物理（physics）等等不一而足。同理，使用 C++ namespace 和全权名（fully qualified name），例如写 gfx::Texture 而不是 gfxTexture。

重要的抽象

将你所有的文件操作抽象出来。在某些平台上 stdio 是不好用的，而且一个清楚的抽象可以让你容易地往引擎里添加新的功能，比如压缩和对文档的支持。你应当尽量减少使用到的磁盘文件的总个数，这样在没有硬盘的游戏机平台上才能获得理想的数据加载时间^①。再者，请不要习惯性地同时打开多个文件，因为光盘的寻道时间是相当长的。可以采取将所有文件一次性读入内存，然后再解析其中内容的方法。

编写你自己的随机数发生器，并一致地到处使用。确保你支持多个随机数流，并清楚地将基于模拟的随机数和基于 camera 的随机数区分开来，以便支持确定的回放模式（replay mode）。这样做，你将在不同平台上得到较一致的结果。

为输入设备增加一层间接操作，禁止较高层次的代码直接检测——不论是“R”键或左摇杆（analog stick）。编写某种 manager 类以提供高层代码检测复位事件或读取刹车脚踏板（brake pedal）的读数的能力。这使得游戏设计师们可以在项目的后期，简单而安全地重映射一些控制键，或支持新的侧面按键个数偏少的游戏机平台。

考虑用你自己的 malloc 和 free 版本来取代原本平台提供的内存管理器。在互联网上可以找到免费而又稳定可靠的实现 [Lea00]。通过使内存管理有更好的一致性，可以确保你新分

^①译者注：俗称读盘时间，越短越好。

配到的内存总是已经初始化成同样已知的状态。

1.4.2 Build 系统

不论你采用什么 build 系统，首先要确保它是容易设置的，并可同时可靠地将设置传达到每个子模块。一般来说，`makefile` 是比较容易管理的。几乎在每一款主流的集成开发环境 (IDE) 或功能较完善的文本编辑器里，都可以用命令行方式调用 `make`，并在一个窗口中依次列出出现的错误。你应当把尽可能多的规则集中在一个公共的 `makefile` 里。惟一应出现在局部模块专属的 `makefile` 里的只有局部模块的名称、组成模块中的库的文件列表，外加零或多个 `use-case` 测试用例。大多数编译器都可以生成自动依赖关系的列表，或你也可以采用免费的 `makedepend` 工具来处理依赖关系。

互联网上可以找到许多各种版本的很好的 `make`。`GNU make` [GNU02] 可能是其中使用最广泛的一个。而较新的工具例如 `jam` [Jam02] 使用起来简单许多。

Build 设置

许多开发工具只能为你的项目提供调试、发行两个版本的 build 设置，但还有一些设置在寻找特定类别的 bug 时很有帮助。下列类别的设置是很有用的。以下各个 build 设置类型之间彼此独立。

- 调试信息，有/无
- 优化，开/关
- 断言 (assertion)，编译/不编译
- 跟踪语句，编译/不编译
- 在游戏内的开发和调节工具，编译/不编译

在许多情况下，选择生成调试信息会使编译和链接的速度显著放慢，因此你并不希望总是生成调试信息。至于打开优化则可能干扰调试，但是显然你会希望在普通版本中打开优化。通过能够独立地控制断言，有利于捕捉到那些由于无意之中增加了有副作用的断言而引起的 bug，特别是当同时还保留很多其他调试信息的时候。

不需要将上面列出的那些 build 设置的所有组合一一定义。其中有一些比较重要，而另一些没什么意思。对每个配置使用一个强制包含头文件 (forced-include file)，以便在需要的时候可以快速地重定义。

你应当把编译环境设置为在容忍范围内尽可能多地生成警告信息。注意应对每个目标平台都这样做（在 `Developer Studio` 里是 `/W4`，在 `gcc` 里则是 `-Wall`，等等）。有些编译器会更挑剔，因此最好将你的代码在尽量多的目标平台上编译，这将会增加在实际运行代码之前就发现一些 bug 的机会。另外，应当总是打开编译器里的“将警告当作错误处理 (Treat warnings as errors)”选项，因为它迫使你保持代码整洁，也有助于避免将可能会在其他平台上产生错误和警告的代码向团队中的其他人发布。

工作室中的每一个项目都应该配备一台专用的 build 计算机。当一名程序员将代码 `check-in` 而向团队中其他成员发布的时候，他应预先登录进入这台 build 专用机，更新代码，将本项目的数个关键设置都 build 一遍，这样做能够迅速发现忘记将新写的程序或修改过的

程序 check-in 的情况，避免其他程序员们因为得到一个不能编译通过的版本而浪费时间。每天深夜时分，这台 build 专用机应当重建 (rebuild) 所有可用的设置，并将所有发现的错误以日志的形式发给开发团队。对于较大的开发团队，不妨连续循环地进行版本重建。

考虑使用外部的源文件验证工具，例如 PC-lint™ [Gimpel03]。强大的 lint 工具将比一般编译器更严格地对代码进行静态语义分析，也能在 build 时发现不少重要的逻辑错误。

1.4.3 细节

上面已介绍了一些有关编写可移植的代码的细节，所以在这里我们只讨论一些容易在游戏中出现的情况。不论是使用你的目标机上可用的中间件 (middleware) 还是本机 API (native API)，增加你自己的优化层都是有益的，特别是因为新的平台或新的中间件总是层出不穷。

1. 数值类型的尺寸

别指望数值类型总是占用固定尺寸的存储空间，也别指望某些类型总是和其他一些数值类型占用一样大的存储空间。请用 typedef 将有符号的和无符号的 8 位、16 位、32 位和 64 位整数定义在你的强制包含头文件中，以便随时使用。

2. 字节顺序

字节顺序 (Byte endianness) 并不像初看上去的那样难对付。其中最大的问题也就是读写在其他平台上生成的二进制数据。选择之一是在你的程序中一致地运用下面这些例程来处理字节顺序的交换：Read/WriteInt、Read/WriteShort、Read/WriteFloat。另一个选择是从根本上避免使用在其他平台上创建的二进制文件。

3. 结构域的顺序

如果可能的话，请把你的结构中的成员域按存储尺寸排序。比方你声明了一个 char，一个 int，然后又是一个 char，编译器可能会插入额外的字节来填充间隙，也可能会完全重新排列结构中各个域的顺序。在 C++ 里，不要忘记父类的数据域，多态类的实例还包含一个指向虚函数表 (v-table) 的指针。当今所有的游戏机都支持某种类型的 SIMD (单指令多数据) 指令集，从而提出了比单个字 (Word) 更严格的对齐要求。在往结构中插入新域的时候，要记得将字节边界对齐的因素考虑进来。必要的话，在运行时打印出那些被频繁实例化的类中每个域的尺寸，确认你了解每个字节的去向。如果出现差异，可以使用 ANSI 标准的 offsetof 宏。

4. 虚函数表指针

各种编译器会在结构中的不同部分插入对象的虚函数表 (v-table) 指针。如果必须把 v-table 指针保存在一起的话，考虑从仅包含一个虚拟析构函数的空类继承，这将保证 v-table 指针在所有平台上都第一个出现。

```
class Base
{
```

```
virtual ~Base() { }  
};
```

在有些平台上，虚拟函数表里的每条函数都缺省地占用 4 个以上的字节。有些旧版 gcc 就饱受这个问题的折磨，其实可以通过 `-fvtbl-thunks=3` 来解决。

5. 断言

断言价值之所在请参考 [Rabin00]。在跨平台类的开发中，由于代码较有可能在未来被一些不求甚解的人所重用，`assert` 是特别有用的。建议你用模版化的数组类来实现所有尺寸固定的数组。虽说使用模版会带来一点额外开销，但是它能通过检测下标越界来避免破坏内存数据，为你节省很多调试时间（以数小时计）。它们在你的数据流水线工具中应有绝对必要的地位，当然也必须尽量友善和可靠。

6. 条件编译

使用条件编译主要理由有二：一是生成同一平台上的有区别的版本。二是生成不同平台上的不同版本。前者是不可避免的麻烦事，而后者可以通过使用类工厂来避免。因为条件编译很快就会使代码可读性大幅降低，所以要尽量限制条件编译的使用量。最好仅在接口中有限制地加以使用，并把平台相关的接口彼此彻底地隔离。当你确实用到条件编译的时候，推荐使用 `#if SYMBOL` 而不是 `#ifdef SYMBOL`。所有基于 gcc 的平台都支持 `-Wundef` 编译开关，可以在发现使用未经定义的预处理符号时产生警告。这可以防止一类由于不小心关掉了一段代码而造成的可恶的 bug。

```
#ifdef XBOX // Typo. code will be silently removed  
...  
#endif
```

7. 预编译的头文件

预编译的头文件是很有用处的。但你需要仔细考虑如何来使用它们。在有些平台上，写一个“`Everything.h`”并在每一个源代码模块的顶部包含，是一个很好的主意。最后你将得到一个单独的大的预编译过的头文件，以后处理起来很快，`build` 耗时也能够极大地降低。这个方法的问题在于，不是所有平台上的全部编译器都支持预编译头文件，一个巨大的头文件在某种意义上也是为缩短编译时间而采取的最差的方法。在实际使用中，该方法使快的平台 `build` 得更快，慢的平台则 `build` 得更慢。不使用预编译头文件是解决办法之一，因为若有设计得当的向前引用，你可以将接口之间的依赖关系最小化。但如果你决心使用预编译头文件，不妨仔细设计一下代码的结构，尽量取其精华去其糟粕。

```
// Class.h  
#if !__PCH  
#include "subclass1.h"  
#include "subclass2.h"  
  
class Subclass3 ...  
  
// Class.cpp
```

```
#if !__PCH
#include "class.h"
#include "subclass4.h"
#else
#include "Everything.h"
#endif
```

如果你不打算采用预编译头文件，那就总是第一个包含源程序对应的那个头文件。这将保证你的类定义不会从之前包含的头文件处引进任何隐藏的依赖关系。

8. 测试类

每一个类都应该具有对应的 use-case 测试用例，简单的测试用例执行库中的函数。它们作为一种形式的文档，也作为一种验证在不同平台上以及在后续的发布版中的功能一致性的手段存在，并有其价值。在库的开发过程中想自己进行验证的话，可以用将输出或屏幕显示捕捉下来，将其与那些已经确定是正确的版本加以比较。

1.4.4 结论

无论你在多大程度上利用中间件，对于今日的游戏开发业来说，设计和开发自己的大型跨平台库依然是极其重要的工作。各大游戏机厂商所占的市场份额并没有一家独大的情况，因此游戏开发商不能只顾其中一个平台而忽略对其竞争对手的主机进行支持。你的前期计划做得越好，对跨平台的维护越用心，你就会在整个过程中遇到越少的问题。

1.4.5 参考文献

[Gimpel03] Gimpel Software, "PC-Lint," available online at www.gimpel.com, March 2003.

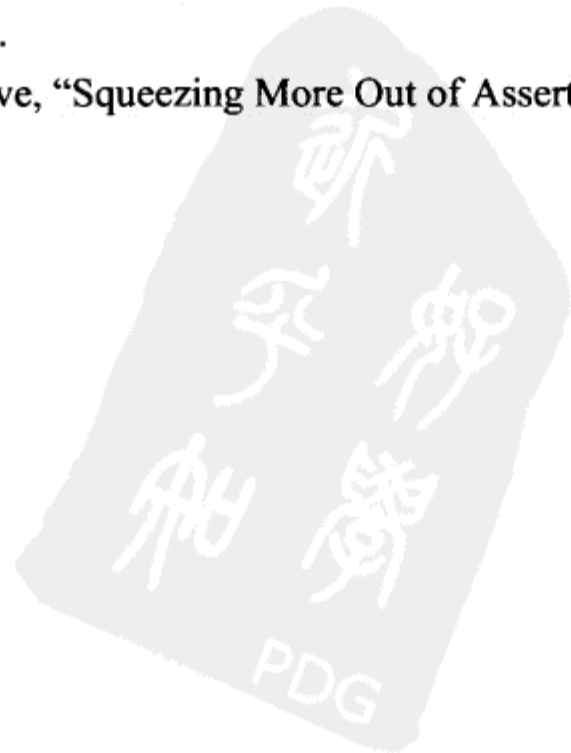
[GNU02] Free Software Foundation, "GNU Make," available online at www.gnu.org/software/make/make.html, April 20, 2002.

[Jam02] Perforce Software Inc., "Jam," available online at www.perforce.com/jam/jam.html, March 2002.

[Lakos96] Lakos, John, *Large-Scale C++ Software Design*, Addison-Wesley Publishing Co., 1996.

[Lea00] Lea, Doug, "A Memory Allocator," available online at <http://g.oswego.edu/dl/html/malloc.html>, April 4, 2000.

[Rabin00] Rabin, Steve, "Squeezing More Out of Assert," *Game Programming Gems*, Charles River Media, 2000.



1.5 利用模板化的空闲块列表克服内存碎片问题

作者：Paul Glinker, Rockstar Games Toronto
E-mail: paul@glinker.com
译者：沙鹰
审校：万太平

在 游戏的世界里，运行时进行的动态内存分配和删除是需要避免的。虽然在某些情况下看来，动态内存操作非常便利，但是它几乎总是会损失速度、导致内存碎片化，最后以较差的访问局部性（locality of reference）告终。但希望仍然是有的。本文将向你介绍模板化的空闲块列表，并展示如何利用该技术，在获得运行时分配与删除的便利的同时又不受相关的负面影响。

1.5.1 内存操作

频繁地进行内存分配与删除可能会造成过多的内存碎片。而且可能导致下面这种情况：有足够的空闲内存可以满足应用程序的申请，但是没有足够大的连续内存块可以满足申请 [Ravenbrook03]。在通常所用的 PC 上遇到这种情况时，我们的每秒帧数立刻会跌入低谷，此时操作系统正试图通过虚拟内存来满足要求。在家用游戏机上问题就更大，由于没有救命的交换文件，内存碎片几乎总能使游戏崩溃（见图 1.5.1）。



图 1.5.1 如图的一个充满碎片的堆中共有 17k 空闲空间，但是每次试图分配多于 7k 都会失败，原因是没有足够大的连续空闲块来满足要求

频繁的分配和删除操作带来的另一个负作用就是低的访问局部性。访问局部性指的是应用程序对邻近的内存位置进行引用操作的方式。一个频繁地引用随机分散在堆中的内存位置的应用程序，具有较差的应用局部性。此类行为会导致缓存错失（cache miss）这个在任何系统体系中都会使性能降低的罪魁祸首，在数据缓存和指令缓存较小的视频游戏机上此症状尤甚。一个持续引用邻近的内存位置的应用程序，具有较好的应用局部性

[Ravenbrook03]。此类行为降低了缓存错失，因而提高了应用程序的性能。频繁地分配和删除将使理想难于实现。

默认的内存管理器是导致性能进一步降低的另一个罪魁祸首。通用的内存管理器不得不在后台替我们解决很多问题。每次我们申请一块内存，内存管理器都要在可用块列表中搜索最佳匹配，若是没有找到，则需要将一块较大的空闲块划分成多个小块，以避免在一次分配中浪费过多空间。当释放一块内存的时候，内存管理器会试图将它与相邻的空闲块合并[Flynn97]以减少碎片。确实，我们希望一个通用的内存管理器替我们进行这些操作，但作为代价而消耗的多个 CPU 运行周期是我们所不希望的。

1.5.2 解决方案

一种既获得运行时分配和删除的便利，同时又避免付出相应代价的方法是使用空闲块列表 freelist。

简单来说，freelist 是可用于分配的内存块的列表，最常见于内存管理器内部。应用程序分配内存时，内存管理器在 freelist 上搜索一块不小于申请尺寸的可用块。应用程序执行内存删除时把可用块归还 freelist。

之前间接提到过，一个功能完善的内存管理器中的 freelist 必须处理一些消耗宝贵的 CPU 周期较多的问题，例如对可变大小的内存块的处理等。为了全面控制，我们将设计我们自己的 freelist，使其独立于默认的内存管理器。实际上，我们将设计数个 freelist，分别针对将在运行时频繁分配和删除的一些不同的数据类型。分配和删除操作直接在这数个 freelist 上进行。例如，一段分配和删除树结点的代码通常是这样写：

```
CTreeNode *pNode = new CTreeNode;  
delete pNode;
```

但现在变成了

```
TFreeList<CTreeNode> TreeNodePool(1024);  
CTreeNode *pNode = TreeNodePool.NewInstance();  
TreeNodePool.FreeInstance(pNode);
```

对于我们的 freelist 的实现，只有几点要求：内存碎片必须消除，访问局部性必须提高，速度必须快，接口必须简单，必须是可重用的且保证类型安全。满足所有这些要求将证明我们之前所付出的不算大的代价是值得的。

1.5.3 实现细节

本文直白的标题以及之前举的例子清楚地显示了我们的 freelist 将是一个模板。之所以采用这个设计决定，除了可重用性外，其他一些好的理由很快也会变得显而易见。我们将模板声明为：

```
template <class FLDataType>
```

当实例化 TFreeList 的时候，首先要做的一件事就是分配一整个数组的 FLDataType 对象，外加一整个数组的 FLDataType 指针。然后将这指针数组作为一个定长的栈来保存指向每一个 FLDataType 对象的指针。

```
TFreeList(int iNumObjects)
{
    ASSERT(iNumObjects > 0);

    m_pObjectData = new FLDataType[iNumObjects];
    m_ppFreeObjects = new FLDataType*[iNumObjects];

    ASSERT(m_pObjectData);
    ASSERT(m_ppFreeObjects);

    m_iNumObjects = iNumObjects;
    m_bFreeOnDestroy = true;

    FreeAll();
}

void FreeAll(void)
{
    int iIndex = (m_iNumObjects-1);

    for (m_iTop = 0; m_iTop < m_iNumObjects; m_iTop++)
    {
        m_ppFreeObjects[m_iTop] =
            &(m_pObjectData[iIndex--]);
    }
}
```

FreeAll()成员函数负责填满指针栈。将此功能与构造函数分开，是因为有时我们需要一下子释放所有东西。

现在，你可能注意到 TFreeList 的构造函数要求 FLDataType 具有默认构造函数，这不一定是方便的。因此，我们有一个能够接受预分配的数据的额外的构造函数。

```
TFreeList(FLDataType *pObjectData,
          FLDataType **ppFreeObjects,
          int iNumObjects)
{
    ASSERT(iNumObjects > 0);

    m_pObjectData = pObjectData;
    m_ppFreeObjects = ppFreeObjects;

    ASSERT(m_pObjectData);
    ASSERT(m_ppFreeObjects);

    m_iNumObjects = iNumObjects;
```

```
    m_bFreeOnDestroy = false;

    FreeAll();
}
```

当申请一个对象实例的时候，TFreeList 会弹出指针栈上的第一个指针，将其返回给我们。

```
FLDataType *NewInstance(void)
{
    ASSERT(m_iTop);
    return m_ppFreeObjects[--m_iTop];
}
```

当释放一个对象实例的时候，TFreeList 会将实例指针归还到指针栈上。

```
void FreeInstance(FLDataType *pInstance)
{
    ASSERT( (pInstance >= &(m_pObjectData[0])) &&
            (pInstance <=
             m_pObjectData[m_iNumObjects-1]));
    ASSERT(m_iTop < m_iNumObjects);
    m_ppFreeObjects[m_iTop++] = pInstance;
}
```

强在何处

我们的实现满足了先前所有的要求，它防止碎片的形成、提高访问局部性、速度快于默认的内存管理器、接口简单、可重用，并且保证类型安全。

通过使用我们存储对象的方法，碎片被完全消除了。我们将指定类型的所有数据预分配在一个连续块中，然后在此块内部进行所有的分配。由于每个单独的块的大小是相同的，连续块的内部不会产生碎片，就算我们频繁地分配和删除对象实例也不会（见图 1.5.2）。

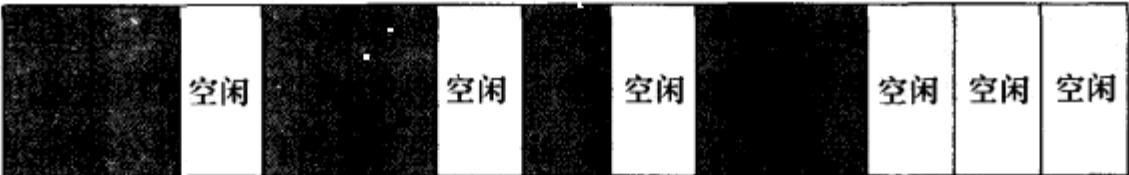


图 1.5.2 即使该连续内存块里的空闲部分是比较分散的，碎片还是不会产生，
因为每次分配和删除的大小是固定的

紧接着就是访问局部性。通过在一个连续块中预分配我们的对象，内存存取被限于内存的同一个区域。按照定义来说，这是对访问局部性的提高，而且应用程序执行大量对同类型对象的连续存取，有助于减少缓存错失的发生。但不幸的是，这项改进对那些具有较差的内存存取模式的应用程序没有作用。

速度的提升部分源于改善的访问局部性，同时也因为我们的 NewInstance() 函数利用了默认内存管理器所没有利用的一点事实：所有块的大小都相同。避免了在内存中进行最佳匹配尺寸块的查找，这无疑给了我们极大的速度优势。我们知道返回的内存块大小总是合适的，因为我们知道 freelist 中每一块的尺寸都是适当的。

确实 freelist 类也可以不用模板实现（利用原始数据），但是模板给我们两个显著优势。第一个优势是，此类模板化迫使编译器保证数据在内存中是正确对齐的。正确地对齐（proper alignment）在 PC 上是很重要的，在游戏主机上则更是至关重要。第二个优势是在编译时间进行的错误检查。当我们错误地试图从一个包含某一类型的 TFreeList 中删除或分配其他类型的数据时，编译器将能够容易地检测出错误的状况。

1.5.4 有效地使用我们的 FreeList

TFreeList 的默认构造函数要求 FLDataType 具有一个默认构造函数，我们可以利用这点。FLDataType 对象的所有一次完成的初始化部分都可以放在默认构造函数中。然后我们可以增加一个 Reset 函数来处理一些小量的、从 TFreeList 分配后需要重新初始化的部分，用法如下：

```
CParticle *pParticle = ParticlePool.NewInstance();  
pParticle->Reset(SMOKE01, orientation, vel);
```

注意事项

将 TFreeList 与容器类（例如二叉树）一同使用的时候，最好让每棵树实例有一个 TFreeList 成员。若我们像这样建立一个全局的树节点池：

```
TFreeList<CTreeNode> g_TreeNodes(MAX_TREENODES);
```

产生多棵树的应用程序到头来还是具有很低的访问局部性。如果把我们的 TFreeList 用作树类的内部成员，内存中分配的物理位置就会更集中。

```
m_pNodePool = new TFreeList<CTreeNode>(iMaxTreeSize);
```

显然，我们的 TFreeList 因为指针栈而增加了额外的内存开销。但这是为了使代码高速、简单、容易让人理解和接受。也可以不使用额外的数据来保存指针栈而实现 freelist 类，在这种情况下，你将用每个空闲实例的头 4 个字节来保存指向下一个空闲实例的指针。但要当心，这样做可能会损坏实例中指向虚函数表的指针（如果有的话，根据你所使用的编译器的实现而定），也就是说你需要在对象上执行多一遍 new 使它变回有效。另外要注意在数据元素的尺寸（字节数）小于 sizeof(FLDataType*) 的时候无法利用该方法。

1.5.5 结论

现在你已经掌握了 freelist 的概念以及我们自行实现的 TFreeList 类，你应当尽可能多地运用这一编程技术。

大多数容器类，包括动态链表、栈、队列、各种树，都可以通过使用 freelist 而受益 [Headington94]。任务管理器 [Harvey02]、粒子系统、玩家角色管理器也可以从中获益。freelist 亦可被用于存储游戏状态对象，从而为应用程序加速一些人工智能中的经典搜索算法 [Russell95]。在网络应用中，或在任何其他资源需要被严格管制的情况下，它也可用作资源管理的机制。

通过阅读本文，你了解了为何在运行时进行动态内存分配和删除会造成问题。避免这些问题的有效途径之一是使用 `freelist`。

如果你已经在游戏中使用了自定义的内存管理器，并还希望进一步提高，可以考虑在你的内存管理系统中结合一个外部 `freelist` 类，并设置其自动向主内存管理器发回统计的跟踪信息。

1.5.6 参考文献

[Flynn97] Flynn, Ida M., and Ann McIver McHoes, *Understanding Operating Systems, Second Edition*, PWS Publishing Company, 1997.

[Harvey02] Harvey, Michael, and Carl S. Marshall, "Scheduling Game Events," *Game Programming Gems 3*, Charles River Media, 2002.

[Headington94] Headington, Mark R., and David D. Riley, *Data Abstraction and Structures Using C++*, D.C. Heath and Company, 1994.

[Ravenbrook03] Ravenbrook Limited, "The Memory Management Reference," available online at www.memorymanagement.org, June 2003.

[Russell95] Russell, Stuart, and Peter Norvig, *Artificial Intelligence, A Modern Approach*, Prentice Hall, 1995.



1.6 一个用 C++ 实现的泛型树容器类

作者: Bill Budge, Electronic Arts

E-mail: billbudge@hotmail.com

译者: 沙鹰

审校: 万太平

树是游戏编程中最重要的数据结构之一, 仅次于数组和表。树可以用来表示角色骨骼 (character skeleton)、场景图 (scene graph)、空间分割 (spatial partition) 及层次包围体 (hierarchical bounding volume)。在素材处理工具中, 树还可以用来表示 shader 或解析后的脚本代码。

要实现一棵经济地使用内存, 同时又能够便捷地遍历和修改的树并不容易。在效率至上的场合, 例如在角色骨骼和分层碰撞检测中, 许多游戏完全避免使用树, 改成预处理到特别的数组中。这样的表示法快速高效, 但是很难修改。这真是太糟糕, 因为动态树可以简单地往角色身上添加附件, 或在游戏过程中更新对象。随着玩家对环境提出更高的交互性要求, 灵活的树结构将更加适用。

树的 C++ 的简单实现有过于缓慢以及浪费内存过多的问题, 并不适合直接在游戏运行时使用 [Kovachev02]、[Peeters03]。但是设计一个快速有效的树的库并不是不可能的任务。本文描述了这样的一个库的设计。

1.6.1 可重用的库

设计可重用的树容器有多种方法。最简单的是“C 语言”风格的方法, 允许树节点的指针为空指针。如果我們是在实现 C 语言的库, 此方法是理所当然的选择。但此方法明显的问题有: 型别安全性 (type safety)、大量的类型转换操作、分别将树本身及其内容在堆上作为单独的对象分配所带来的开销。

“经典的” C++ 方法通过继承实现树: 定义一个节点类 `node` 以及一个包含节点的树类 `tree`。使用者从作为基类的 `node` 类型继承出可添加到树中的新节点类型。可是这并没有解决型别安全性的问题, 且同样需要大量的向下转型操作 (downcasting)。使用者还须对每个新类型的树定义新的节点类。

“现代的” C++ 方法以类属的方式实现容器。创建一个模版类用于对象类型为 `T` 的树。C++ 标准模板库 (STL, Standard Template Library) 是该方法最好的实例。泛型库 (generic library) 使用灵活, 效率又高。以下我们就参照 STL 的例子来设计一个泛型的树的库。

1.6.2 树的概念

树是节点的集合，像自然界中的树那样，节点之间以枝状结构彼此连接。按照惯例，树是从上往下生长的（见图 1.6.1）。一个节点下面可以挂着零或多个子节点，同时子节点又可以是某棵树的根节点，因此树是一种递归结构。挂着子节点的节点称作内部节点（interior node）或分支节点（branch node），没有子节点则称为叶节点（leaf node）。树最顶端的节点称为根节点（root）。在名著《计算机程序设计艺术》的第一卷基本算法 [Knuth73] 中有关于树的全面讨论。

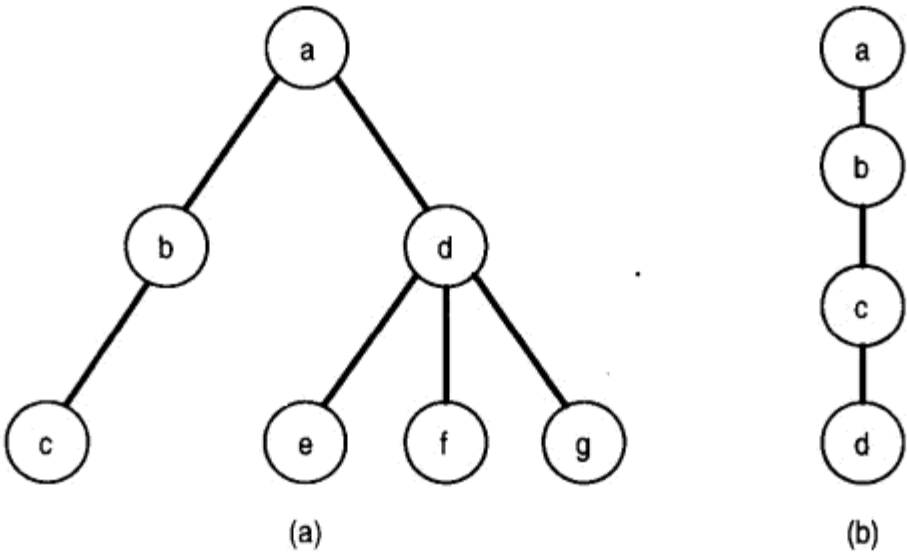


图 1.6.1 树的例子

有几种不同的方法可以遍历树上的内容。前序遍历对于每个节点，先访问节点本身，然后访问子节点（a-b-c-d-e-f-g 是如图 1.6.1a 的树的前序遍历）。后序遍历对于每个节点，先访问子节点，然后访问节点本身（c-b-e-f-g-d-a）。层次序遍历对于每个节点，都先访问节点本身，然后访问子节点，接着才访问子节点的子节点，并以此类推（a-b-d-c-e-f-g）。在游戏编程中最重要的遍历序是前序。这既是骨骼动画系统计算变换的自然顺序，又是在视锥（view frustum）中剔除场景图的自然顺序，也是寻找与分级的边界域发生交叉的自然顺序。后序遍历在解析脚本和 shader 代码的时候很有用，因为只有在处理了所有子节点后才能处理父节点。层次序遍历较不常用。因此我们的设计原则之一是让前序遍历的速度尽量快。

在大多数运用树结构的场合，树中任意节点的子节点之间没有顺序关系，这种树称为无序树，我们可以将前序遍历的结果颠倒来模拟后序遍历的结果。若树中任意节点的子节点之间有顺序关系，这种树称为有序树。此时可以通过将子节点以逆序加入树中而实现后序遍历的效果。在本文中，我们将实现正向和反向的前序遍历。

1.6.3 树的实现

实现树节点的简单方法之一是使用列表或动态矢量来保存其子节点。这是为了方便执行

修改操作，例如增加或删除节点。遍历也很简单。下面给出将某棵树中的内容以前序遍历输出到一个流中的代码。

```
struct tree
{
    T _value;
    std::vector<tree> _children;
};

void OutputTree(tree& t, Stream& output)
{
    output << t._value;
    int childCount = t._children.size();
    for (int i = 0; i < childCount; i++)
        OutputTree(t._children[i], output);
}
```

该方法有一个大问题就是浪费内存。有效的动态矢量实现将平均浪费所分配内存的 25%~33%。虽然可以用链表代替矢量，但是链表上通常带有一个哑元节点 (dummy node)。因此无论采用那种方法，树上每个节点仍然有一定的内存浪费。

此设计的另一个问题就是，我们不得不编写递归函数来对树进行遍历。递归优美但效率不高。就算能通过将递归用明确的栈和循环改写而减少函数的作业开销，遍历树操作仍然是昂贵的。不论采用何种遍历方法，每当在树上进行额外的操作时，代码就不得不重复一遍。这很无聊，代码也不好维护。若我们打算在实现中从矢量改为使用列表呢？那将会带来极大的麻烦，因为每个处理函数中的 for 循环都只能改写了。较好的设计应能将遍历逻辑限制在库的内部。

幸运的是，我们仍能进一步提高树的效率。如下所示的 node 结构仅使用了两个指针域，但已足以表示一棵树。

```
struct node
{
    T _value;
    node* _first_child;
    node* _next_sibling;
};
```

要访问某个节点的子节点时，取其 _first_child 指针域，然后沿着取出的第一个子节点的 _next_sibling 域，直到遇到 NULL 值为止。可是，这样做并未解决速度问题。仍然需要一个栈来遍历树，而且树修改函数的实现也将遇到严重的困难。我们可以通过增设更多指针来解决所有这些问题。当每个节点采用 4 个指针域时，我们能够高效率地进行前序遍历，同时可以高效率地修改树。

```
struct node
{
    T _value;
    node* _parent;           // 若是根节点则值为 NULL
    node* _prev_sibling;     // 指向前一个兄弟节点
```



```

node* _next;           // 指向前序下的下一个节点
node* _last_descendant; // 若是叶节点则指向自身
};

```

`_next` 指针使前序遍历快如闪电，只要一重间接引用即可。但后序遍历怎么办呢？不存在 `_prev` 指针，事实上也不需要一个什么 `_prev` 指针。无论树的尺寸如何，总能由 `prev()` 函数在常数时间内计算出结果。

```

node* prev() const
{
    node* prev = NULL;
    if (_parent) // 树的树根没有前趋
    {
        if (_parent->_next == this) // 是第一个子节点吗？
            prev = _parent;
        else
            prev = _prev_sibling->_last_descendant;
    }
    return prev;
}

```

下列在树中导航的工具函数也能在常数时间内执行。

```

node* first_child() const
{
    node* child = NULL;
    if (_next && (_next->_parent == this))
        child = _next;
    return child;
}

```

```

node* last_child() const
{
    node* child = first_child();
    if (child)
        child = child->_prev_sibling;
    return child;
}

```

```

node* next_sibling() const
{
    node* sib = _last_descendant->_next;
    if (sib && (sib->_parent != _parent))
        sib = NULL;
    return sib;
}

```

```

node* prev_sibling() const
{
    node* sib = NULL;

```

```

    if (_parent && (_parent->_next != this))
        sib = _prev_sibling;
    return sib;
}

```

本设计的诀窍之一是：除了第一个节点外，任何节点的 `_prev_sibling` 指针总指向前一个兄弟节点，这一直进行到最后一个子节点，从而形成了环形单链表。为了在常数时间内取得某节点的最后一个子节点，这一点“小手脚”是必要的。

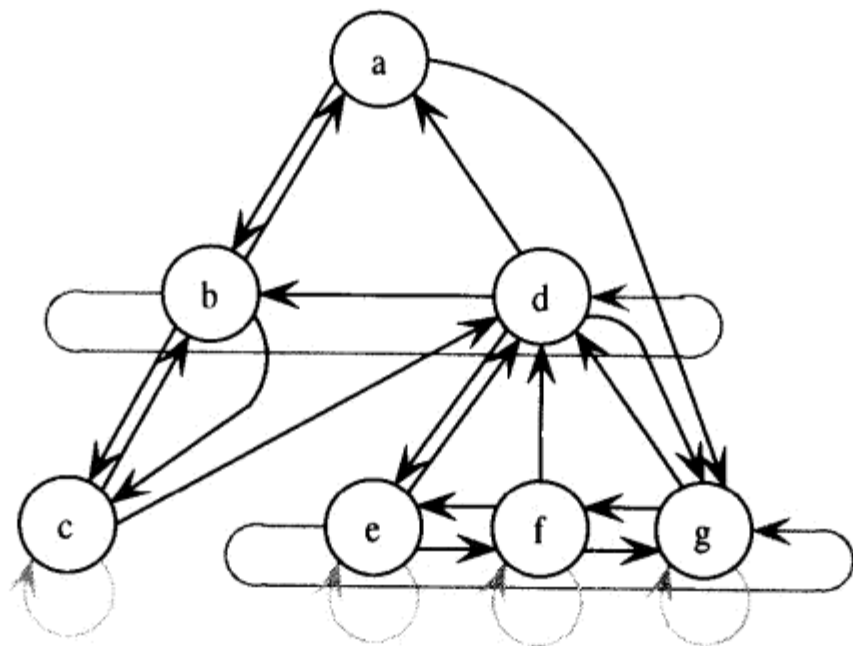


图 1.6.2 树的指针图

有了上面那些导航函数，就可以着手编写用来修改树的函数了。我们将所有对树的修改实现为如下两个树的成员函数。

```

void insert_subtree(TreeT& child, TreeT* next);
void remove_subtree(TreeT& child);

```

采用 `_last_descendant` 指针所带来的重要影响是，往树里增加一个节点可能会造成上溯至根节点的节点改动。事实上，这会在通过前序添加新子节点构造树的时候发生。在最坏的情况下（图 1.6.1b），代价与树中元素数量的平方成正比。在实际使用中，因为最坏的情况尤其罕见，也因为通常树在初始时建立完毕之后只会偶尔进行修改，这不算一个大问题。无论何时，若这成了问题，用户可以将子节点以相反的顺序插入树中。

1.6.4 利用 STL

已经明白如何实现树以后，就到了设计接口的时候了。我们将采取 STL 模式，既为了熟悉，也为了让我们尽可能利用 STL 代码。那么我们能使树看上去像其他 STL 容器一样吗？

STL 容器为了实现顺序访问容器中的内容同时并不暴露内部表示，提供了迭代器 (iterator)。这概念就是，允许 STL 将容器和算法之间的耦合解除。算法可以与任何提供迭代器的容器协同工作。如果我们采用相同的方式对 `tree` 类进行封装，将节点隐藏并通过迭代器引用 T 型对象，那么代码可以这样写：

```
tree<T> t;
tree<T>::iterator it;
for (it = t.begin(); it != t.end() ++it) {}

// 利用 <algorithm>标准函数库头文件
void MyTFunction(const T& t);
for_each(t.begin(), t.end(), MyTFunction);
```

不巧的是，由于迭代器仅仅提供了对 T 对象的访问，我们同时也隐藏了树的层次结构。当处理树的时候，通常有必要对节点的双亲节点或孩子节点进行引用。在我们之前的例子里，是无法获得此类相对关系的。那么对迭代器增加一个成员函数来获得临近的节点如何呢？这样做将解决第一个例子中的问题，但仍不能解决第二个例子中的问题，因此不是个好主意。迭代器应尽量与指针相似，也就是说不要为迭代器编写成员函数 [Alexandrescu02]。

问题在于，STL 容器的概念并不适合树结构。树并非是 T 型对象的线性序列，而是一个包含 T 型对象的层级结构。因此我们要大胆地丢下 STL 容器的概念，强调树的内部结构。树是子树的线性序列，而树的迭代器提供的是对子树的访问。

如何取得 T 型对象呢？方法是通过 tree 的一个公有数据成员 value 来访问。由于它属于使用者，将此成员封装起来是没有意义的事。因此我们可以编写如下的代码：

```
tree<T> t;
for (tree<T>::iterator it=t.begin(); it!=t.end() ++it)
{
    tree<T> subtree& child = *it;
    if (!child.is_root())
        child.value *= child.parent()->value;
}

void MyTreeFunction(tree<T>& t);
for_each(t.begin(), t.end(), MyTreeFunction);
```

实际上，若无需下降到子树中就能对树中的子节点进行迭代，也是很有用的。为此，可以简单地提供 child_iterator。我们可有效率地（常数时间复杂度的增一和减一操作）用 next_sibling() 和 prev_sibling() 函数实现 child_iterator。iterator 与 child_iterator 都是双向的，定义了增一和减一操作。最后，我们可以利用一定的 STL 机制，用不多的几行样板代码来实现反向迭代器、反向子节点迭代器。

给定了树的遍历函数，实现迭代器的惟一困难就在于怎样处理终点的值，以便能从终点往回迭代。在终点处增设哑元节点，对于 std::list 那样的 STL 容器还能解决方法，但是对于树不管用。要设就要在每个树节点上都设一个哑元节点，这显然是不实际的。因此，我们的迭代器保存两个指针，一个指向当前树，另一个指向产生自身的树根。迭代进行到终点的时候，当前指针设为 NULL。减一操作若检测到 NULL 的存在，便会将当前指针指向树根下的最后一个后继。换句话说，这样的迭代器和列表迭代器是类似的，在自身指向的子树被删除前一直保持有效。

这样我们就了解了 tree 类的接口应是什么样的。我们定义一些简单的构造函数和析构函数。同时我们再定义拷贝构造函数和赋值运算符，使树具有完整的值语义。这两个函数执行

树的完全拷贝功能，提供对树结构的强大的构造能力。

```
tree();
tree(const T& t);
tree(const TreeT& copy);
~tree();
const tree& operator=(const TreeT& rhs);
```

由于我们的树的实现是一个子树的线性序列，感觉很像列表。因此，模仿 `std::list` 的接口是很自然的事情。

```
void push_back(const TreeT& subtree);
void push_front(const TreeT& subtree);
void pop_back();
void pop_front();
iterator insert(iterator it, const TreeT& subtree);
iterator erase(iterator it);
void clear();
iterator splice(iterator it,
               iterator first, iterator last);
iterator splice(iterator it,
               child_iterator first, child_iterator last);
```

其中 `splice` 函数（接合函数）通过修改内部指针而移动树节点的位置。这比起先复制子树然后再删除原始子树效率要高许多。

注意，每个接受 `tree` 类型作为参数的函数同时也能接受 `T` 类型的参数，这是因为我们定义了构造函数 `tree(const T& t)`，因而不需要显式类型转换的缘故。考虑到 `tree<T>` 和 `T` 之间存在着自然的一一映射关系，在此允许隐式类型转换罕有地没有带来任何问题。

最后，让我们定义一些树特有的操作。

```
bool is_root() const;
bool is_leaf() const;
bool is_descendant_of(const TreeT& ancestor);
size_t size() const; // 后代的总数
size_t degree() const; // 直接子节点的数量（出度）
size_t level() const; // 在树中的深度
```

STL 的原则之一是异常安全（exception safety），这意味着容器类在其内部 `T` 型对象抛出某些异常的时候必须保证不被修改。虽然大多数的游戏都避免异常的使用，但其实要实现异常安全并不需要费太大力气。树容器可以用 [Sutter00] 中描述的技术来满足这条 STL 保证，本身不抛出任何异常。

内存分配总是游戏程序员难于处理的问题。作为一个适合在游戏中使用的库，必须允许用户自行控制内存的分配行为。STL 容器模版通过接受内存分配器 `allocator` 作为可选参数之一，允许用户的代码可在必要时自行定义内存分配行为，因此正好满足要求。我们也会提供这样的灵活性，但是又带来了一个问题。树递归地包含子树这样优雅的概念使我们没有余地来保存一个 `allocator` 对象。解决方案可以是将树和节点作为不同的概念，将 `allocator` 保存在树中。不过，由于不可能脱离 `tree` 对象对树节点进行操作，`tree` 类用起来有一点点不

方便。

若不要求太高，我们可以在不放弃优雅的递归树概念的同时，获得使用 `allocator` 的大多数好处。方法是将 `allocator` 作为 `tree` 类的静态成员。但这限制了对同一批模板实例而成的树只能使用同一个 `allocator`，也表明了 STL 标准使实现者难于提供更多的灵活性 [Plauser01]。而现在，有了合适的自定义 `allocator` [Isensee02]，`tree` 类可以不寻常地在运行时代码中得到大量重用。

1.6.5 结论

我们所设计的树的库可生成能容纳任意类型的对象的树。若是担心模板代码膨胀 (template code bloat)，可以仅实例化 `tree<void*>`，并对型别安全的树的指针接口进行私有继承，从而获得一棵型别安全的 C 语言风格的树 [Meyers98]。模版化的容器是强大且灵活的。

这一个树的库提供强大的构造树的功能。可以通过一个与 `std::list` 相仿的简单接口往树中添加类型为 `T` 的对象。由于 `tree` 类有值语义（一个拷贝构造函数和赋值运算符），我们可以用一行赋值语句来复制整棵树。为了效率，`splice` 函数更能够将数棵子树“粘贴”起来形成一棵大树。

树构造完毕后，可以快速地进行正向或反向的前序遍历，甚至还可以用子节点迭代器进行递归遍历。

```
// 前序遍历，使用递归
void DoSomething(tree<T>& t)
{
    t.value.DoSomething();
    tree<T>::child_iterator it;
    for (it=t.begin_child(); it!=t.end_child() ++it)
        DoSomething(*it);
}

// 后序遍历，使用递归
void DoSomething(tree<T>& t)
{
    tree<T>::reverse_child_iterator it;
    for (it=t.rbegin_child(); it!=t.rend_child() ++it)
        DoSomething(*it);
    t.value.DoSomething();
}
```



参见附带光盘上使用 `tree<T>` 的例程源码。

1.6.6 参考文献

[Alexandrescu01] Alexandrescu, Andrei, *Modern C++ Design*, Addison-Wesley, 2001.

[Isensee02] Isensee, Pete, "Custom STL Allocators," *Game Programming Gems 3*, Charles River Media, 2002.

[Knuth73] Knuth, Donald, *The Art of Computer Programming: Fundamental Algorithms*, Addison-Wesley, 1973.

[Kovachev02] Kovachev, Alexander, "Tree data class for C++," available online at www.codeproject.com/cpp/treedata_class.asp, January 13, 2002.

[Meyers98] Meyers, Scott, *Effective C++*, Addison-Wesley, 1998.

[Peeters03] Peeters, Kasper, "tree.hh: An STL-like C++ Tree Class," available online at www.damtp.cam.ac.uk/user/kp229/tree/, April 17, 2003.

[Plauger01] Plauger, P. J., et al., *The C++ Standard Template Library*, Prentice Hall, 2001.

[Sutter00] Sutter, Herb, *Exceptional C++*, Addison-Wesley, 2000.



1.7 弱引用和空对象

作者: Noel Llopis, Day 1 Studios

E-mail: llopis@convexhull.com

译者: 沙鹰

审校: 刘永静

程序员常常对指针又爱又恨。指针占用很少的存储空间,而且可以很有效率地处理,用户通过指针可以真正自由地进行内存操作。但是,在指针问题上,哪怕是最小的错误也常会导致内存泄漏,或者在最坏的情况下带来程序崩溃和用户的崩溃。

本文关注一些处理游戏资源的时候,由在内存中进行对象移动操作而导致的常见问题。在此讨论的技术也将与关于载入和卸载内存对象的问题密切相关。我们将看到弱引用是如何成为直接使用 C++ 指针之外的一个好的选择。作为对弱引用的补充,我们将用到空对象,从此再也不必在每次使用对象前检查指针是否为空了,而且这还将带来一些有价值的附加作用。

1.7.1 使用指针

虽然指针是很方便的,而且我们也对指针使用有着多年的经验,但是对于有些任务,指针并不是最好的方法。让我们举操作游戏中的动态资源为例。在这里的资源一词,是指所有从磁盘读入的数据:纹理贴图、场景模型、声音等等。

我们在处理指针时遇到的根本问题就是确定资源的生命周期。如果你载入一张大贴图,同时有两个模型共享这同一张贴图,它们一定是指向内存中的同一个位置。那么在其中一个模型被删除的时候应怎么办呢?肯定不能删除贴图,因另一个模型还用着它。但是这贴图怎能知道是否还有其他对象在引用自身呢?肯定我们也不能不进行处理,因为要是那样,等两个模型都被删除的时候,这个贴图就是一大块内存泄漏。

在这样的情况下,我们必须提出自己的方案。通行的做法是进行引用计数(reference counting)。也就是说每当有一个模型获得了一个指向某张纹理的指针,该纹理的引用计数器就加一。相应地,每当一个模型被删除时,它原先引用着的纹理的引用计数器就减一。当引用计数器的值降到零时,就自动删除该纹理。在理论上,这是个很好的主意,但它倾向于产生臃肿的代码,常常成为编程错误的发源地。

还有相关的问题，比如在内存中移动资源，或只是暂时地将资源卸出内存而旋即载回。如果那张贴图仅仅被用在游戏世界的一小部分里，那么我们何不当玩家身处其他地方的时候将其卸载，当玩家再次接近这个区域时再将其载入？这样做将节省很可观的内存数量，从而使我们能够突破目标开发平台上的内存限制，也不再局限于只能设计小规模的游戏。

我们如何将贴图交换出去呢？方法之一是通知程序中拥有该贴图指针的部分，使其暂停使用该贴图。要不然当它们试图使用该贴图的时候，程序多半会崩溃。即使这张贴图一直没有显示出来的机会，但等到重新读入这张贴图的时候，该贴图很可能会保存在一个不同的内存位置里。为了避免该问题，我们需要一个方法来更新所有使用该贴图的对象里指针的值。

理想的解决方案应该给我们这样的功能，即对离开镜头很远的模型载入中等程度的、仅包含一些 mipmap 的贴图，而对距离镜头很近的那些模型载入最清晰的、也是占用内存最多的贴图版本。

和许多其他计算机科学领域里的两难问题一样，该问题可以通过增加一层间接引用来解决。使用句柄是可行的方法之一。每一个资源都由叫做句柄的唯一的 ID 来标示（可能在载入时由资源管理系统指定）。游戏本身承诺决不保存指向资源的指针，而是保存句柄。每一帧，每当游戏需要显示一个模型或进行任何需要访问该贴图的计算时，应要求资源管理器返回对应该句柄的贴图指针。然后我们检查返回的指针是否是空的（NULL），如果非空，则加以处理。

这给了我们卸载资源，或改变它们在内存中的位置的自由。资源管理器能够跟踪响应的改动，而程序的其他部分总是通过句柄间接地获得指针来进行操作。不幸的是，这很不方便：每次都要从句柄转换成指针，还非得检测得到的指针是否为空不可，很快就让人厌烦了。而且还有一个问题，如果我们决定将一个指针保存“几帧而已”，可是资源在这几帧里卸载了又会怎样？

下面将介绍一个解决方案，在提供直接操纵指针的舒适感和自由度的同时，解决了之前我们提到的大多数问题。

1.7.2 弱引用

从本文的角度来看，弱引用指的是那种和实际指向的对象之间至少存在一层间接的引用。例如之前提到的句柄，因为通过映射机制实现了一层间接，就是一个很好的弱引用的例子。而基本的 C 语言指针，因为采用直接的引用模式，则是一种强引用（指针包含所引用的对象的实际内存地址）。

注意“弱引用”这一个名词通常在关于内存管理的文献中有另外的含义。在内存管理中，缺乏强引用指向的对象将被系统释放，因此弱引用并不强制保留对象。

我们的目的是使弱引用像指针一样容易使用。由于其行为像指针，但内嵌的逻辑会处理间接引用的本质，这一类引用通常称作智能指针（smart pointer）。

可以用基于句柄的系统来构造智能指针 [Hawkins02]，但在本文中我们将采取一种稍微不同的方法。为了追求效率和简单性，也为了容易进行调试，我们将通过包裹一个原始 C++ 指针来实现智能指针。

图 1.7.1 是该系统的组织结构 UML 图。我们的智能指针 ResourcePtr 并不直接指向资源，而是指向一个中间对象 ResPtrHolder。该中间对象含有一个指向资源的普通指针。资源管理器通过使用 map 容器将资源名字与资源指针的 holder 联系在一起，从此每当一个资源需要被更新或 invalidate 的时候，holder 中的内容会自动被修改，而游戏中所有的智能指针将自动地仍使用正确的值。

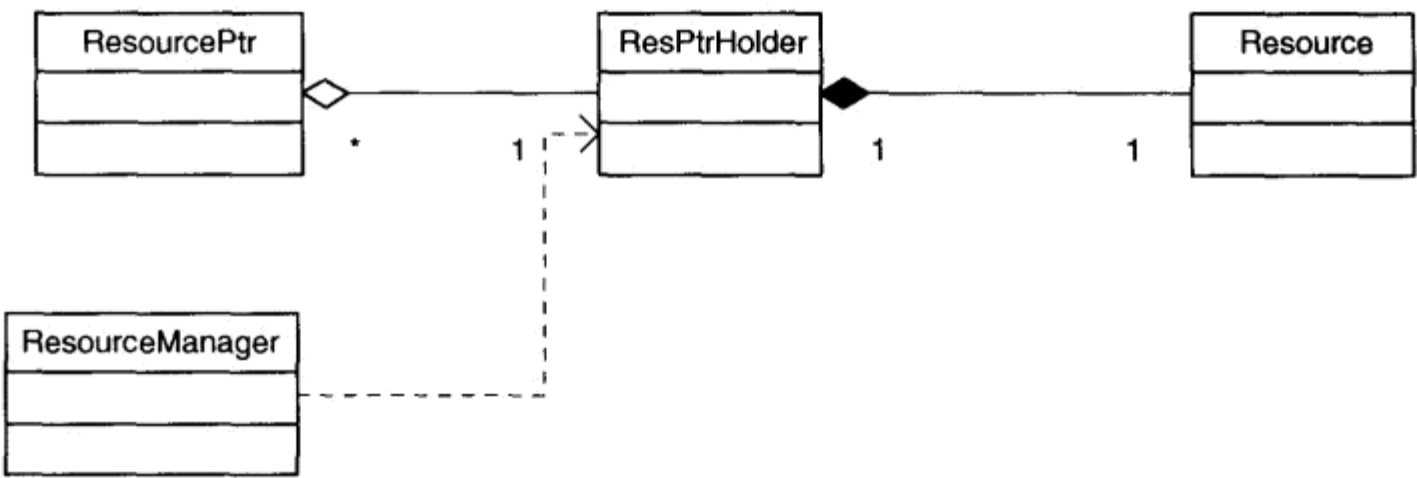


图 1.7.1 资源系统里的类结构

1. 智能指针

要怎样才能构造出看上去像指针，行为也像指针，而且内置了额外逻辑的智能指针呢？答案很简单，特别是无需考虑由对象生命周期的确定、或写时拷贝语义（Copy-on-write Semantics）而带来的问题的话。我们只要重载一些运算符就能使它看起来像指针，例如重载 operator->和 operator*。也不妨将智能指针实现为模板类，以便实现型别安全。你可以在 [Meyers96] 和 [Alexandrescu01] 中找到更多关于智能指针的阐述。



下面给出让人最感兴趣的部分，我们的智能指针的接口定义。随书光盘里含有完整的源程序，还有一些单元测试集。请注意在程序里我们是如何使用运算符来避免直接访问，而通过指针 holder 间接访问数据的。

```
template <class ResType>
class ResourcePtr
{
public:
    ResourcePtr(const ResourcePtr<ResType> & ResourcePtr);
    explicit ResourcePtr(ResType * pRes);
    ResourcePtr(ResPtrHolder * pHolder = NULL);
    ~ResourcePtr();

    ResourcePtr<ResType> & operator= (
        const ResourcePtr<ResType> & ResourcePtr);

    ResType * operator->() const
    {
        return static_cast<ResType *>
            (m_pResHolder->pRes);
    }
}
```

```
ResType & operator*() const
{
    return *(static_cast<ResType *>(
        (m_pResHolder->pRes)));
}

bool operator==(ResourcePtr<ResType> res) const;
bool operator!=(ResourcePtr<ResType> res) const;
bool operator< (ResourcePtr<ResType> res) const;

private:
    ResPtrHolder * m_pResHolder;
};
```

关于智能指针的一个重要方面就是它与普通指针占用同样大小的存储空间，也就是说在大多数平台上是 32 位。这意味着我们可以高效率地复制智能指针，将智能指针当作参数来传递，或对智能指针进行任何可对普通指针进行的处理。此外，本文中实现的智能指针是基于指针而非句柄的，因此并没有多少效率的损失。不需要在每次访问资源时进行句柄的查找。事实上，惟一的代价就是要对指针解引用。虽然有代价，但代价并不大，考虑到智能指针的作用，付出这一点点性能的代价是值得的。

2. 资源指针 Holder (ResPtrHolder)

资源指针 Holder 类看上去十分简单。它要做的只是保存一个指向资源的指针而已。每当需要新的 Holder 对象时，必须进行动态分配。不能把 Holder 对象保存在数组或矢量中，是因为我们必须能在不修改其他 Holder 对象的内存位置的情况下进行创建和删除操作。进行多次小规模的内部分配可能会浪费许多内存，因此，最好是重载 Holder 类的 new 和 delete 操作符，并确定是高效地分配自内存池 (Memory Pool) [Llopis03]、[Glinker04]。这样就不会占用多余的存储空间，运行时效率非常好。

值得注意的是，尽管你很容易觉得它是一个模板类，ResPtrHolder 本身不是一个模板。虽然把 ResPtrHolder 做成一个 ResType 类型实例化的模板可能是个好主意，也可以减少在 ResourcePtr 类里进行难看的静态类型转换，但是做成模板的话，只是将同样的转型操作移到资源管理器类中进行罢了。事实上，将这些转型推给资源管理器类会带来一系列额外的复杂问题，尤其是当你用一个全局资源管理器来管理所有资源的时候。



ON THE CD

现在我们已经是在常规的指针机制之下隐藏了额外的一层间接，然后我们可向资源指针 Holder 类添加引用计数来解决资源的生命周期问题。智能指针对象的创建和销毁操作将控制 Holder 中的引用数，每当没有引用剩余，Holder 便删除资源并删除自身，当然还要通知资源管理器。本书的光盘里包含另一个版本使用了引用计数的智能指针系统。

3. 资源管理器

资源管理器其实就是一个大 map，其中保存了资源名字和资源指针 Holder 之间的关系。

注意这里使用资源指针 Holder 而非资源本身，这样一来所有的操作都在 Holder 上进行，也就不担心接下来在游戏中会破坏智能指针了。

根据游戏的架构不同，你可能会为游戏中的所有资源使用同一个全局的资源管理器，也可能为每一类资源使用专用的资源管理器。后者的优点是可以清楚地了解所处理的是何种类型的资源，有较多具体信息，因此比较容易采用不同的算法来进行资源回收和交换。

这个 map 可以有多种方法来实现。直接的做法是采用成份是字符串和资源指针 Holder 的 `std::map`。在实际游戏编程中，你一定希望这映射要尽量高效，因此你可以改写映射函数，也可以用整数作为键。你可以在读取数据的时候，用资源名称通过散列（hash）或循环冗余校验（CRC）方法得到整数键值，也可以对每个资源指定一个独特的资源 ID。若你愿意为较高的访问效率而多牺牲一点内存，可以使用 `std::hash_map`，将元素访问时间降到常数水平。

1.7.3 空对象

弱引用的概念很棒。我们终于可以无忧无虑地卸载资源或将资源在内存中搬移了。不过，我们还有一个问题尚未解决：必须检查指针是否为空（NULL）。

1. 概念

因为我们已经那么的习惯于检查 NULL 指针（通过条件判断语句，或使用 `assert`），我们常常并不考虑太多。可是，这使代码变得臃肿，代码的本义因为源程序中充斥的大量条件判断语句而变得晦涩。而且从技术上讲，尽管不算严重，这些条件判断还是对程序效率造成负面影响的。

如果可以假定资源指针总是有效的不是很好吗？程序就再也不用检查指针是不是 NULL 的了。如果资源一经读入后，就永久地保留在内存里，这就容易做到。但是如果资源是动态载入的，就不太容易了。

这就需要用到空对象。空对象也是资源，不过是没有任何数据的，或者是为了能够像正常资源一样动作而仅拥有最低限度的数据。例如对于贴图资源，空资源可以是一张很小的 4×4 大小的完全透明的贴图。这个概念适用于其他任意类型的资源：模型（一个只有一个顶点的 mesh）、声音（一段很短的沉默）、动画（静止）等等。

每当决定要从内存中回收某个资源时，首先释放资源对象本身，但接下来并不将资源指针 Holder 中的指针设成 NULL，而是将它指向一个相应类型的空对象。如此一来，程序能够自由的访问任何资源，而不用担心碰到空指针。每个资源都保证在内存中，只不过有可能是个空对象罢了。

之所以尽可能地让空对象不可见，是为了避免在程序试图显示空对象的时候造成不愉快。例如，玩家角色跑到了一个新地区，此时原本在远处的一块街道指示牌应当被显示在屏幕上，但是这牌子上的贴图还没有被读入进来。使用空对象可以保证这指示牌在贴图读入完毕之前都是透明的，直到读入完毕时突然显示出来。当然我们希望此时指示牌仍然在足够远的地方，那样突然显示出来也不会太显眼。不过一般而言，你可以把系统设计成具有平滑地

混合 (Blend) 空对象和已载入的对象的对应显示的功能。

为了空对象能正常工作, 我们需要为每类资源保存一个空资源。如此, 空资源的行为就能和其他同类资源在所有方面保持一致。比方说程序访问一张贴图时, 可能会试图在这个贴图资源上调用贴图专用的函数, 空对象也具有这些函数因此不用担心。

2. 潜在的问题

处理空对象的时候, 要注意以下事项。

第一点, 如果程序要从资源中取得数据, 那么最终的行为不太可能正确。举例来说, 如果程序试图访问一块已经予以回收的场景纹理贴图, 会得到一块非常小的空贴图对象。但是, 由于游戏中所有的场景纹理贴图的尺寸都是 128×128 , 程序也假定这块贴图的尺寸也是 128×128 , 从而访问到了空贴图对象中并不存在的像素。理想地, 程序不应该访问已经回收的资源, 因为那样做会带来一个 bug (稍后详细介绍)。但若是程序无法避免访问已回收的资源, 就要特别注意, 在随时可能对资源进行交换的系统中不应对资源作任何假设。

第二点是关于试图修改资源的程序的。幸运的是, 这个情形并不多见。大多数资源都是只读 (Read-Only) 的, 因为这样可以共享资源。不过, 遇到必须修改时我们必须有所准备。也许我们是想在贴图上烫上几个金字, 或是想随时随地的修改场景模型。试图在空资源对象上执行上面那些操作会带来不好的后果, 例如导致所有的透明空对象突然变成亮红色。为了避免问题, 将空资源标记成只读是有益无害的。或者更进一步, 若我们要维持全透明, 可以允许程序通过调用函数来修改空资源, 不过一定要确保这些函数对资源本身没有影响。

3. 扩展

至此我们已经将空对象整合进了我们的资源管理系统, 接下来可以更进一步, 使它们更有用处。例如, 可以规定永远不要把空对象显示出来, 显示出来就是 bug。在这个情况下, 我们会希望空对象越显眼越好。空贴图可以用亮粉红色, 声音可以用惹人厌的响亮的方波声音, 场景几何体则用一个大球体表示, 总之是可以立刻吸引人的注意力的那种。而且, 若是游戏试图显示空对象, 可以往游戏日志文件里加一条信息, 注明游戏试图显示的是哪一个资源。

不过, 在发行版本里, 我们可能希望恢复之前的行为, 使空对象尽量隐形, 以防玩家在最终发售的游戏里看到瑕疵。

1.7.4 结论

本文对使用指针来记录游戏资源介绍了一种灵活的替代方案。新方案使我们对资源的生命周期有更好的控制, 也允许我们自由移动资源。甚至可以在游戏运行时将一些资源从内存中卸出。所有这些新功能将使我们能够开发更大的游戏关卡, 创造连续的游戏世界。

本文也介绍了空对象的概念, 也就是用来替代那些不在内存中的资源的特殊资源。这些特殊资源的行为和其他同类资源相同, 因此游戏代码无需在每次使用资源的时候检查该资源是否已经载入。空资源一般是看不见的, 以避免被显示出来而造成的瑕疵, 不过在调试阶段也可以让空资源醒目地显示。

1.7.5 参考文献

[Alexandrescu01] Alexandrescu, Andrei, *Modern C++ Design*, Addison-Wesley, 2001.

[Glinker04] Glinker, Paul, “Fight Memory Fragmentation with a Templated Freelist,” *Game Programming Gems 4*, Charles River Media, 2004.

[Hawkins02] Hawkins, Brian, “Handle-Based Smart Pointers,” *Game Programming Gems 3*, Charles River Media, 2002.

[Llopis03] Llopis, Noel, *C++ for Game Programmers*, Charles River Media, 2003.

[Meyers96] Meyers, Scott, *More Effective C++*, Addison-Wesley, 1996.



1.8 游戏中的实体管理系统

作者: Matthew Harmon, eV Interactive Corporation

E-mail: matt@matthewharmon.com

译者: 万太平

审校: 肖丹

新入门的游戏开发者们常常会被自己的劳动成果给迷倒: 开发一个新的 shader 程序, 创建一个粒子系统或者播放 3D 的音效。虽然这些层面的开发是很重要, 而且的确有趣, 然而使一堆松散的代码变成完整功能的游戏就常常不是那么迷人的任务了。本篇文章则把焦点集中在此游戏实体的管理上。

为了阐明实体管理的原理, 而不仅是展现一个具体的实现, 本文提供一个 C 语言的解决方案。由于用 C 语言编写的代码采用了基于对象、有多态特性和良好封装的实现方法, 这个系统转换到 C++、Java 或者 C#都非常容易。

1.8.1 概述

现代游戏充满各种各样的实体。玩家、敌人和发射的炮弹, 这些要素在游戏世界里乱哄哄地跑来跑去。地形、建筑物、天空和云彩定义环境。路径点 (waypoint)、触发器 (trigger) 和脚本指导玩家的体验。得分、破坏和物理方面组成游戏世界的逻辑和规则。

不是对于所有的这些东西分别对待, 作为特殊目的的要素, 把它们结合为一个系统并且提供一个共用的结构和通信方法是很方便的。基于消息的方法来管理实体能够解决很多问题并提供一种方法来统一大部分的关键游戏要素。

1. 任何事情通过消息来出现

在一个基于消息的系统中, 实体做的任何事情是对应消息的。为了画出一个实体, 你发送出一条消息。为了让实体在空间移动, 你则给它也发送一条消息。假如它们都没有取得任何消息, 实体会无所事事而不做任何事情。

Win32 的 windowing API 就是个很好的比方。Windows GUI 做的任何事情是对应消息的, 它们从应用程序、其他窗体 (windows) 或者操作系统而来。实体, 像窗体一样对于它内部数据或者实现一无所知。它们简单地发送和响应定义好的消息集。实体对于它们不关心或不能理解的消息则不予理睬。

2. 一个简短的例子

用一个简单的例子来深入研究系统是很有用的。让我们用玩家发射一个飞弹作为例子。因为玩家和飞弹两者都是实体，这个操作是一个简单的消息通信问题。实际上，这一代码能够用于从任何实体类型中发射所有类型的发射物。

```
ENTITY* FireProjectile(  
    char*   className, // 待生成的发射物 (Projectile) 的类型  
    ENTITY* shooter,   // 发射器/所有者  
    ENTITY* parent)    // 场景图中的父节点  
{  
    ENTITY* entProj;    // 发射物实体  
    VECTOR3 pos;        // 发射器的位置  
    VECTOR3 dir;        // 朝向矢量  
    VECTOR3 velocity;   // 初速矢量  
  
    // 将发射物作为场景图中父节点下的子节点而创建  
    entProj = EntCreate(className, parent, 0, 0);  
    if (entProj)  
    {  
        // 将发射物作为场景图中父节点下的子节点而创建  
        EntSendMessage(entShooter, EM_GETPOS,  
                        (int)&pos, 0);  
        EntSendMessage(entShooter, EM_GETDIR,  
                        (int)&dir, 0);  
        EntSendMessage(entShooter, EM_GETVELOCITY,  
                        (int)&vel, 0);  
  
        // 设置新发射物  
        EntSendMessage(entProj, EM_SETPOS,  
                        (int)&pos, 0);  
        EntSendMessage(entProj, EM_SETDIR,  
                        (int)&dir, 0);  
        EntSendMessage(entProj, EM_SETVELOCITY,  
                        (int)&vel, 0);  
        EntSendMessage(entProj, EM_SETOWNER,  
                        (int)entShooter, 0);  
        EntSendMessage(entProj, EM_START, 0, 0);  
    }  
  
    return(entProj);  
}
```

3. 万事万物都是实体

构建实体管理系统中的一个关键方面，就是将任何事物都作为能通过消息来控制的实体来考虑。试图统一像发射物、地形和游戏脚本这样多样的要素，虽然看起来有点不可理解，但是通过统一，我们得到了系统的强大和简单性。

就像一颗子弹需要更新位置、检查碰撞以及被画出，场景需要被画出并响应碰撞检测。一个脚本测量一关的“赢的条件”需要在每一帧执行，但是不对任何消息响应。从这个简单的例子，我们能够开始通过消息统一我们的实体。表 1.8.1 说明了三个不同的实体类型是怎样处理各种消息的。

表 1.8.1	子弹、地形和脚本实体怎样响应各种消息		
	子 弹	地 形	脚 本
更新	处理	忽略	处理
画	处理	处理	忽略
检查碰撞	处理	忽略	忽略
响应碰撞检测	忽略	处理	忽略

在这个方案，即使每个逻辑脚本在世界中的物理意义上不存在，也能够像其他任何实体那样对待。这个实体机制能够在更加复杂的游戏代码周围充当一个简单但强大的封装层。

4. 系统的要素

实体管理系统由 4 个主要的组件组成。

- 实体消息：定义实体怎样沟通。
- 实体代码：实现一实体类的代码和数据。
- 类列表：维护一个已经注册在案的实体类的列表。
- 实体管理器：创建实体、管理实体树、支持发送消息到一个或者多个实体。

1.8.2 实体消息

消息本应被看做函数调用——对一个实体做一个请求（或者要求）去做某事。大部分的实体仅仅需要对你定义的消息集合中的一部分做出响应。因此，实体必须确实能够忽略它们不需要或者不理解的消息。

一些消息，如 DESTROY，是简单而无需任何其他数据的。但其他如 SETPOSITION 的消息则需要参数。在下面的 Win32 例子中，我们支持两个普通参数命名为 var1 和 var2。它们被声明为整型数，但是它们实际的类型则是依据被发送的消息来定义的。在 SETPOS 情况，var1 参数用于传递 3D 向量的地址。假定这样，对于我们标准实体的消息处理函数的原型定义如下：

```
// 消息处理函数的类型
typedef int(ENT_PROC)(
    ENTITY* ent,      // 指向 _this_ 这个实体容器
    EM    message,    // EM... 实体消息
    int    var1,       // 通用参数 1
    int    var2);      // 通用参数 2
```

在一个 C++实现中，参数 var1 和 var2 的类型转换消失有利于虚函数取正确的类型参数。函数的返回值发信号是否应该对整棵树继续递归。假如希望，允许实体树作为场景图(scene graph)。

你创建的消息形成所有的游戏实体操作的协定。消息列表应该好好公布并归档。


```

typedef enum entMessageTag
{
    // 类操作
    EM_CLSINIT,    // 初始化类
                  // 数据路径(char*)通过 var1 传入
    EM_CLSFREE,    // 释放类
    EM_CLSNAME,    // 将类的名字复制到 var1 中

    // 创建和销毁
    EM_CREATE,     // 创建实体
    EM_START,      // 启动实体
    EM_SHUTDOWN,   // 完整地销毁实体
    EM_DESTROY,    // 立即销毁实体

    // 标准动作
    EM_UPDATE,     // 经过的时间以秒为单位通过 var1 传入
    EM_DRAW,       // _normal_ 渲染

    // 数据访问
    EM_SETPOS,     // 位置 (VECTOR3*) 通过 var1 传入
    EM_GETPOS,     // 将位置 (VECTOR3) 复制到 var1 中
    ...
} EM;

```

增加新的消息就像添加另外的值到这个枚举类型中那样简单。因为老的实体将简单忽略新的消息，系统能无需破坏现存的代码来获得扩展。

1.8.3 实体代码

在 C 中，实体本质上是一个数据类型和单一处理消息的函数。另外，可能有静态类数据，比如资源句柄 (resource handle)、游戏性平衡性调整值等。对于飞弹 (missile) 的基本模式 (忽略功能内容) 可能看起来像这样：

```

// 飞弹类
typedef struct missileTag structure
{
    char        name[MAX_NAME];
    VECTOR3     velocity;    // 速度矢量
    VECTOR3     position;    // 全局位置
    VECTOR3     forceAccum;  // 所受合力
    MATRIX4     matModel;    // 朝向矩阵
} MISSILE;

// 通过 CLSINIT 载入，并通过 CLSFREE
// 释放的资源及其他多个类利用的变量
static MODEL   mdlMissile;
static SOUND   launchSound;
static float   thrust = 10000.0f;

```

```
// 消息处理函数
int MissileProc(
    ENTITY*    entity,    // 实体容器
    EM         message,    // 经处理的消息
    int        var1,      // 通用参数 1
    int        var2)      // 通用参数 2
{
    MISSILE* e; // 指向实体数据的指针

    // 从容器中获取实体的类数据
    e = ((MISSILE*)entity->data);

    // 处理一切与飞弹实体有关的消息
    switch(message)
    {
        // 类操作
        case EM_CLSNAME:
            strcpy((char*)var1, "MISSILE");
            return(TRUE);
        case EM_CLSINIT:
            return( ClsInit((char*)var1) );
        case EM_CLSFREE:
            return( ClsFree() );

        // 创建和销毁
        case EM_CREATE:
            return( Create(ent) );
        case EM_SHUTDOWN: // 对于飞弹来源, shutdown 和 destroy 是一样的
        case EM_DESTROY:
            return( Destroy(e) );
        case EM_START:
            return( Start(e, var1, var2) );

        // 标准动作
        case EM_UPDATE:
            return( Update(e, var1) );
        case EM_DRAW:
            return( Draw(e) );

        // 数据访问
        case EM_SETPOS:
            V3Copy(&e->position, (VECTOR3*)var1);
            return(TRUE);
        case EM_SETVEL:
            V3Copy(&e->velocity, (VECTOR3*)var1);
            return(TRUE);
        case EM_SETDIR:
            return(SetDirection(e, (VECTOR3*)var1));
        default:
    }
```

```
        return(DefEntityProc(message, var1,
                               var2));
    }
    return(TRUE);
}
```

对于消息处理器 (handler) 的第一个参数是一个普通的实体容器用于引用系统中任何实体类型。在本文后面会详细描述。在提取容器的私有数据后, handler 向量屏蔽它们的处理函数。可提供一个 default case 以支持原始的形式, 类似 Win32 中的 DefWindowProc 方法。

1.8.4 类的代码

对于外面的世界, 一个实体类仅仅公开它的消息处理函数, 非常类似 Win32 的窗口类。这个类列表维持一个链表或者这些消息 handler 的向量, 允许类通过一个文本的名字来引用和创建。这个类结构非常简单。

```
typedef struct entityClass
{
    char          name[64]; // 独一无二的类名
    ENT_PROC*     clsProc;  // 类的消息处理函数
    struct entityClass* next; // 表中的下一个类
} ENTCLASS;
```

函数 EntCreateClass 完成注册。其主要功能是增加一个入口到类列表和发送一条 EM_CLSNAME 消息给 clsProc, 目的是取回与这个类相关的文本名字。然后它发送 EM_CLSINIT 消息给 clsProc, 给这个类一个机会去完成只需一次的初始化, 比如装载资源等等。一个游戏中使用的各种类初始化能够在启动时用类似的代码来完成:

```
void GameInitClasses(
    char*  dataPath) // 实体加载数据的路径
{
    EntCreateClass(PlayerProc, dataPath, 0);
    EntCreateClass(MissileProc, dataPath, 0);
    ...
}
```

另外, EntDestroyClass 和 EntDestroyAllClasses 这两个函数将 EM_CLSFREE 消息发送给一个或全部注册类, 使得清除过程变得容易。

1.8.5 实体管理器

实体管理器 (entity manager) 创建和摧毁单个的实体和管理列表或者包含它们的树。实体通过一个简单的、普通的“容器 (container)” 使用指针引用实体消息处理函数和它私有的数据。是这个共同的容器结构允许系统对于所有的实体一视同仁。另外, 还有域 (field) 支持树或者列表, 也支持惟一的 id (guid 数据成员) 用于在一个网络内使实体同步。

```
typedef struct entityTag
{
    ENT_PROC*      Proc;           // 消息处理函数
    void*          data;           // 实体的数据
    int            guid;           // 特有的 id
    struct entityTag *parent;       // 父节点或 NULL
    struct entityTag *prevSibling;  // 前一个兄弟节点或 NULL
    struct entityTag *nextSibling;  // 下一个兄弟节点或 NULL
    struct entityTag *child;        // 第一个子节点
} ENTITY;
```

实体管理器关键操作是：

```
ENTITY* EntCreateEntity(char *className,
                        ENTITY *parent,
                        int var1,
                        int var2);
```

这个函数在类列表中查找 `className`，假如找到，创建一个新的 ENTITY 类型，把它作为父节点的孩子节点挂到实体树中。然后它保存一个指针到类的消息 `Proc`，立即使用参数 `var1` 和 `var2` 发送 `Proc` 一个 `EM_CREATE` 消息。在 `CREATE handler` 中，实体为类数据和节点数据分配空间给它。

```
int EntDestroyEntity(ENTITY* ent);
```

通过实体内部调用响应 `DESTROY` 消息，这个释放 ENTITY 容器结构并且从实体树上解下来。为了实际摧毁一个实体，只要简单发送它一个 `EM_DESTROY` 消息。

```
int EntSendMessage(ENTITY *ent,
                  EM message,
                  int var1,
                  int var2);
```

使用特定的参数发送一个消息给单个实体。

```
int EntSendMessageGuid(int guid,
                      EM message,
                      int var1,
                      int var2);
```

通过 `guid` 而不是实体指针发送一个消息给单个实体。这个对于网络同步很有用处，而这里指针没有任何意义。

```
void EntSendMessagePre(ENTITY *ent,
                      EM message,
                      int var1,
                      int var2);
```

按前序遍历的顺序发送一个消息给实体和所有它的孩子节点。当任何实体在处理一条消息返回 `FALSE` 后放弃递归。若将场景图或者碰撞树也用实体树表示，这一做法就可以支持分层的剔除算法 (`hierarchical culling`)。不会中断 (`abort`) 的类似函数也同样有用。

一个强健的实体管理器也将支持通过名字找到实体，重排实体树，遍历所有实体并对每个实体调用回调函数（callback）。

1.8.6 基于消息的游戏循环

虽然实体能够（且必将）彼此直接通信，主循环负责发送重要的消息实际使得游戏发生。因为几乎所有的“游戏代码”最初停留在实体本身上面，循环变成主要的消息分派者。

```
void GameProcessInput()
{
    GetInputEvents(&inputEvent);
    MapInputEventsToGameEvents(&inputEvent,
                               &gameEvent);
    EntSendMessage(entPlayer, EM_USERINPUT,
                  (int)&gameEvent, 0);
}

void GameUpdateWorld()
{
    EntSendMessagePre(entWorld, EM_UPDATE,
                     elapsedMs, 0);
    EntSendMessagePre(entWorld, EM_POSTUPDATE, 0, 0);
}

void GameDraw()
{
    // 让所有实体先渲染完毕，然后才轮到 overlay
    EntSendMessagePre(entWorld, EM_DRAW, 0, 0);
    EntSendMessagePre(entWorld, EM_DRAWSHADOW, 0, 0);
    EntSendMessagePre(entWorld, EM_DRAWOVERLAY, 0, 0);

    // 若是处于 debug 模式，则给出实体的额外信息
    if (debugMode)
        EntSendMessagePre(entWorld, EM_DRAWDEBUG,
                          0, 0);

    // 形象地将当前选中的实体表示出来
    if (editMode)
        EntSendMessage(entBeingEdited, EM_DRAWEDIT,
                        0, 0);
}
```

1.8.7 开始：消息类

实体的每个类需要初始化和关闭。这些消息对每个实体进程发送一次就够了，因为它们应用到整个类，而不是实体的一个单独实例。当游戏开始或者在装载每一关的时候，类能够被初始化，这个要因游戏而异。

EM_CLSINIT

这个消息一旦发送给游戏中的每个实体类。在接收到它时，类能够从磁盘装载它们的资源，设定类范围的“静态 (static)”变量，和可能创建一个提供给游戏内部编辑的参数符号表。对于 CLASS_CREATE 的一个普通参数可能是直接到应用程序数据的一条路径。

EM_CLSFREE

这个给一个类有机会释放它装载的所有资源和分配的任何内存。类可能在游戏关闭或者需要从某一关退出的时候被摧毁。

EM_CLSNAME

返回类的名字。这个使用类的名字提供类列并支持一个实体需要知道另外和它通信的实体类的罕见情况。

1.8.8 从小处着手：基本实体消息

在最简单的游戏中，实体需要被创建、摧毁、更新和画出。

EM_CREATE

在其容器通过实体管理器创建一个实体时，实体立即收到 CREATE 消息。为了响应这个消息，实体分配它需要的内存给它内部的数据和初始化任何默认的值。这个类似调用一个对象的构造器，而且的确可以用那种方法实现。

EM_SET/GET_POS, _DIR, _VELOCITY, _YAW, _COLOR, 等等

这些“accessor”消息用于设定和取回实体数据。一旦创建，实体通常发送几个这样的消息到位置和在世界中为它定位。

EM_START

在实体被创建、定位或设定后，它发送 START 消息。对应实体能够播放一个初始的声音或者触发一个初始的动画状态，否则当响应 CREATE 的时候可能不会被做。

EM_DESTROY

这个消息需要立即摧毁一个实体。像一个对象析构器 (destructor)，DESTROY 给一个实体机会整理自己。这个包括释放内存、摧毁任何它创建的孩子节点。DESTROY 典型用于仅仅在当玩家玩完一关或者关闭游戏的时候。为了优雅地摧毁实体，需要调用 SHUTDOWN。

EM_SHUTDOWN

有时一个实体需要“关闭”，而不是立即摧毁。例如，一燃烧的火可能需要被“关闭”，但保留的烟雾缭绕一个允许随着时间流失而驱散。在这种情况下，SHUTDOWN 告诉实体“通

知制造任何新的烟雾，当最后的烟雾被驱散后就 DESTROY 自身。”在通常的游戏过程中，与 DESTROY 相比，优先调用 SHUTDOWN。当然，对于很多的实体类型，SHUTDOWN 和 DESTROY 将执行相同的代码。

EM_DRAW

实体通过在世界中渲染它们自己来响应这条消息。这是一个“正常的”游戏模式的渲染。其他类型的渲染在后面讨论。

EM_UPDATE

这个消息给一个实体机会更新在世界中的位置。在游戏中使用可变的时间更新，逝去的时间自从上一帧将被作为一个参数传递给这条消息。固定帧速率游戏在合适的中断频率时可能出现 UPDATE 消息问题。

EM_POSTUPDATE

像碰撞检测和一些 AI 中，某个操作需要明白所有实体已经被更新为后世界中的“新(new)”状态。在所有实体更新操作完成后，典型发送像 POSTUPDATE 的消息。

EM_USERINPUT

这个消息用于发送输入设备事件或者状态更新给当前玩家控制的实体。在 debug 模式，将很快改变那个接收用户输入数据的实体。

1.8.9 游戏和环境消息

消息也能用于游戏环境方面的消息交流，也实现重要的游戏性构想。

EM_FORCE

一个爆炸的实体能够广播 FORCE 消息给整个世界。实体响应计算在 FORCE 消息中的效果和积累结果到它们的力向量中。

EM_DAMAGE

当一个发射投射 (projectile) 实体击中一个目标，它发送一条 DAMAGE 消息告诉被击中者，它需要告知有多少点被破坏。

EM_GIVEPOINTS

当一个实体被摧毁，它会授予破坏它的投射实体的主人以点数，因而在实体自身内部实现一完整的积分系统。

1.8.10 系统成长：一些高级消息

这些消息说明增加较高级的功能怎样变得容易。高级特性能够无需破坏旧的实体而增加

到新的实体；它们简单地忽略新的消息。

EM_DRAWOVERLAY

某个实体在所有“普通(normal)”渲染操作做完之后，可能需要执行画(drawing)操作。例如，当玩家看着它的时候，太阳实体可能处理画炫目的光或者光环。在发送“普通(normal)”DRAW消息后，游戏循环能够发送DRAWOVERLAY给所有的实体。

EM_DRAWSHADOW

类似，很多阴影算法在主要渲染之后被执行。DRAWSHADOW给实体渲染它们阴影特效的机会。

EM_SETOWNER

假如实体树被作为场景图，则SETOWNER能够被用于构建实体之间的所有权关系。例如，当玩家实体发射一个火箭，它发送SETOWNER给新的火箭，让火箭知道是谁开火的。

EM_IMDEAD

宣布一个实体即将来临的破坏常常是重要的。例如，两个精灵(elve)跟踪同一个兽人(orc)。其中一个精灵攻击并且消灭了兽人，而另外一个仍然在跟踪它。目标兽人在它的DESTROY handler发送一条IMDEAD消息给整个世界。这通知第二个精灵是寻找一个新目标的时候了。这避免了持续对于空实体指针的检查。

下面的例子说明，刚刚讨论的消息的一个接合怎样用于支持破坏、积分和“死亡通知(death notifications)。”它在DAMAGE handler中发生。

```
// 处理 EM_DAMAGE 消息
static DamageHandler(
    ENTITY* me,
    ENTITY* sender, // var1 = 导致 damage 的实体
    int hitVal) // var2 = 造成伤害大小 (HP)
{
    // 消息 sender 的 owner (若是发射物)
    ENTITY* owner=NULL;

    // 削减自身的体力，判断是否已经死亡
    me->hitPoints -= hitVal;
    if (me->hitPoints < 0)
    {
        // 我已经挂了！取出 sender 的 owner (如果有的话)
        EntSendMessage(sender, EM_GETOWNER,
            (int)&owner, 0);

        // 若有 owner 的话，一定是一个发射物
        // 因此为 owner 加上相应的分数
        // 否则为 sender 加上相应的分数
        if (owner)
            EntSendMessage(owner, EM_GIVEPOINTS,
```



```
        me->points, 0);  
    else  
        EntSendMessage(sender, EM_GIVEPOINTS,  
            me->points, 0);  
  
    // 告诉全世界我已经挂了  
    EntSendMessagePre(world, EM_IMDEAD, 0, 0);  
  
    // 自我销毁  
    EntSendMessage(me, EM_SHUTDOWN, 0, 0);  
}  
}
```

1.8.11 处理碰撞

处理碰撞可能是游戏中最复杂的任务之一。在缺乏统一的碰撞检测系统时，消息能够再一次来救援。下面的例子稍微有点简单化，但说明了消息能够怎样成长来支持复杂的交互。

EM_TESTHIT

这条消息请求接受者执行“打击检查 (hit test)。”因为很多碰撞类型需要被询问，它通常需要传递一碰撞测试结构来描述执行的测试类型。例如，对于一颗子弹执行可能的碰撞 ray 相对多边形 (ray versus polygon test) 的测试是足够的。然而，巨大的交通工具可能需要请求一体积相对域体积的测试 (volume versus volume test)。

在任何情况，接受者负责执行测试和返回测试结果给调用者。假如需要，允许每个实体类以习惯的方式处理碰撞。随同实际的物理碰撞，这条消息用于支持 3D 挑拣和视线测试。

EM_IHITYOU

当一个实体通过一条或者多条 EM_TESTHIT 消息确定它是实际与一对象碰撞时，EM_IHITYOU 消息被发送给碰撞者，消息的内容包括碰撞本身和实体做出的响应。

1.8.12 扩展到多玩家

使用几条额外的消息，实体体系结构也能够扩展处理网络游戏的要求。在这个系统，实体能够使用很少需要从高级游戏代码那的帮助来在网络之间同步。使用这个系统，实体能够完全独立与实际的网络传输机制。它们简单地发送和接收消息。

EM_SERVERUPDATE

服务器节点发送 SERVERUPDATE 处理 AI、决策 (decision-making) 和位置更新 (position updating)。另外，在服务器上的实体能够传输网络包穿过节点它们本身来同步。

EM_CLIENTUPDATE

这条消息不是用于标准的 EM_UPDATE，而是允许客户端代码仅仅在未知状态推测 (dead-reckon) 它们的实体的，无需执行任何 AI 或者动力学 (dynamics)。

EM_NETPROCESS

当游戏循环接收到网络流量 (network traffic)，它通过 NETPROCESS 向多个实体方向分发数据包。允许每个实体类用它自己本身最有效率的网络同步方法。在某种意义上，服务器上的实体和它本身远处的版本构建直接通信。

1.8.13 开发和调试消息

仅在游戏开发、调试 (debugging) 或者游戏编辑 (editing) 中支持特殊的消息是很有帮助的。下面是一些常见的例子。

EM_DEBUGDRAW

实体响应这个消息通过画附加的调试信息。这 and 把实体名字画在游戏内的 3D 表示上面一样简单。

EM_EDITDRAW

这条消息给实体画出另外的信息的能力，可能当一个游戏在“编辑 (edit)” 模式下是很有用的。例如假定，一个格斗模拟有一个雷达装置能够察觉玩家。雷达站响应 EDITDRAW 通过渲染一线框模式的球体来表示雷达能够获取目标的范围。

EM_GETVARIABLE

使用这条消息，实体能够暴露内部变量的表给编辑的对话框，允许游戏为每个类创建一个定制的编辑工具。

1.8.14 好处

基于消息的实体系统的实际好处很多：

- **同质性：**各不相同的游戏实体能够使用相同的系统来管理和控制，即隐藏更加复杂的代码到精简型实体 wrapper。
- **功能分权：**帮助加强渲染、动力学、逻辑甚至客户端相对服务器端任务的分离。这对于大的项目至关重要。
- **单控制点：**实体的所有扩展头式通过单控制点， EntSendMessage 指令。因此，消息日志、DLL 接口和挂靠系统到脚本语言就变得容易。
- **完全抽象：**发射一颗子弹和发射一飞弹的过程是相同的。只是改变类的名字和发送相同的消息。简化后的类将忽略更多兄弟姐妹类需要的消息。
- **可扩展性：**增加新的特性就常常如创建一个新消息那么简单。因为老的实体能够忽略新的指令，增加的特性不会打破什么。旧的代码能够调上来在后面加速。
- **有限的依赖性：**在一个纯消息驱动的系统，只有消息列表（或者基类）需要用每个实体模块来包含。

- **重用**：正确地构建，实体能够无需做任何改变就可以重用到新游戏中。FPS 中的天空实体能够在飞行模拟器中工作得很好，只要支持正确的消息。

1.8.15 光盘中的内容



在附带光盘中包含的代码大致描绘了一个用 C 语言实现的实体管理系统的轮廓。它并不是让你直接使用的，而是提供了一个框架，在上面能够构建一个完整的包，并能设计 C++、C# 或者 Java 的实现。在这个 C 实现中的类型转换 `typecasting`、`switch` 声明和函数指针看起来有点过时，但它们表明，一个强大的实体系统能够用任何语言创建。

移植到 C++

当用支持类和虚函数的语言进行编程时，基于消息的实体系统将变得更加简单且容易维护。实体类实现从强健基类中继承必需的函数。发送直接的消息就是调用一个成员函数的简单问题。发送一消息给整个实体树通常更加复杂，但易于管理。

当创建一个实体可能就如使用基础类中的新运算符那样简单，维持一个类注册/实体工厂（`class registry/entity factory`）是很有用处的，这样实体能够通过文本名字来创建。这一做法降低了对源文件的依赖，并且允许实体彼此互相创建和操作，而无需了解其中任何类的定义。

1.8.16 总结

创建一个强健的实体管理系统是一个游戏程序员能够承担的最有价值的任务之一。它提供了一个全局框架，游戏剩余的部分可以适应它。更重要的是，它创建了一系列标准和协定，方便快速地原型化和开发。实现系统和对于老的系统花样翻新需要时间，但相比增加新特性来说有上百倍的好处。



1.9 Windows 和 Xbox 平台上地址空间受控的动态数组

作者: Matt Pritchard, Ensemble Studios

E-mail: mpritchard@ensemblestudios.com

译者: 沙鹰

审校: 刘永静

大多数游戏程序员们都常常会遇到管理动态数组(尺寸动态增大或缩小的数组)的任务。我们所采取来处理这一例行任务的直接算法实现, 多年以来一直没有大的改变。如今, 许多程序员会编写类或模版使数组的基本管理操作自动化。本文将为这一原始而基本的做法提出一个相对较新的改进, 旨在为处理大型数组或高度活跃(highly active)的数组, 提供一些虽不甚明显但却实实在在的性能提升。

1.9.1 传统的动态数组管理

传统数组管理的标准做法, 一般不外乎以下几点。

(1) 为所有私有数组变量, 保存以下特征值:

- 数组中的单个元素占存储空间的多少, 以字节为单位;
- 当前所允许的数组中元素的个数上限;
- 当前游戏正在使用的数组元素的个数;
- 一个指针, 指向一块动态分配而得的、足以容纳最多个数的数组元素的内存块。

(2) 当程序需要往数组里增加一个元素时, 需要执行以下步骤。

• 如果当前在使用的元素个数小于当前允许的元素个数上限, 则返回一个指向数组中第 n 个元素的指针(n 等于当前使用中的元素个数加上 1)。不然, 则数组首先增加其长度, 然后再执行本步骤。

(3) 为了增加数组长度, 需要执行以下步骤:

a. 让数组元素的个数上限增加, 可以是增加一个固定数, 也可以是一个有算法推导而得的量(例如将当前长度加倍);

b. 分配一块足够大的存储空间, 以便容纳新的上限个数个元素;

c. 已经被游戏使用的数组元素将被复制到新缓冲区中;

d. 旧的数组缓冲区予以回收, 数组指针指向新分配的存储空间。

(4) 在数组中删除个别元素, 一般不需要进行缓冲区缩小操作。当整个

数组被直接清空的时候，则一般要回收数组缓冲区，并可能需重新初始化。

大多数有经验的程序员们都很熟悉以上描述的方法。从性能提高的角度看，惟一有意义的就是数组增大的算法。合理选取每次数组增大以容纳新元素的尺寸增加值是很关键的。通过就特定问题相关的信息进行全盘考虑，该值可以被调整和优化，达到尽可能降低数组长度变化操作执行次数的目的。



在本书配套光盘上的 `ArrayManager.h` 和 `ArrayManager.cpp` 这两个文件中，定义了 `CArrayManager` 类，也就是以上方法的一个典型 C++ 实现。该类的 `Initialization()` 和 `AddItem()` 方法直接实现了上述操作。注意，这里提供的 `CArrayManager` 类只是为了揭示本文所提及的基本机制和框架，因此为了避免混淆，没有提供那些非必要的功能。`CArrayItem` 元素代表那些由调用者进行分配，并以引用类型传入 `CArrayManager` 类（由其产生一份自身数据的拷贝）的结构或类实例。在一个较为复杂的实现中，可能会放弃使用 `CArrayItem` 类，而是依靠数据类型模版化。

1.9.2 深入观察

在上面一节里描述的方法是最直接的方法，而且似乎没有什么提高的余地。如果我们只是考虑算法和用来实现的高级语言本身，也许是没有什​​么余地的。但，若我们退一步做全盘考虑，就会发现另一个直接影响效率的因素——操作系统。正是操作系统本身，通过一个特定语言的运行时库（runtime library），为编译后的代码提供平台上的内存操作服务。

当代的视窗操作系统，也包括 Xbox 游戏机，为程序做了两件好事：一是提供了一片私有的地址空间，这样每个程序都觉得自己是内存中当前仅有的程序；二是提供了一块虚拟内存（通常用硬盘上的交换文件来实现），允许程序使用比实际物理内存容量更大的存储空间。就算像 Xbox 那样没有分页文件，虚拟内存也能够允许程序成功地申请并使用具有连续地址的存储空间，而不论可用的物理内存里是否有足够大的连续空闲内存块。通过这些新的内存管理手段，我们能够进一步提高已久经考验的数组管理算法的性能。

注意，本文以下将要讨论的技术主要是针对 PC 机和 Xbox 上的 32 位 Windows API 平台的。许多其他的游戏机硬件平台，特别是 PlayStation 2 和 Gamecube，并没有能够支持虚拟内存的内存管理硬件，或其上运行的操作系统并不将对地址空间的控制暴露给应用程序。因此，本文介绍的技术可被认为是 Win32 上的专用技术。

1.9.3 地址空间管理 != 存储管理

当一个 PC 或 Xbox 程序通过 `new` 或 `malloc` 来调用内存分配服务的时候，程序语言的运行时库首先尝试从程序自身拥有的私有堆中寻找匹配的空闲内存来满足分配的需求。如果申请分配的内存实在过大，或因为堆中的数据碎片太多，它会向操作系统申请更多存储空间。当内存分配的要求传达到操作系统后，操作系统会进行以下两项操作。

操作系统做的第一件事，就是在维护地址范围的列表中查找一个足够容纳申请尺寸的空

闲地址范围。在 Windows 里，程序能够访问的总地址空间约是 2GB 不到，因为 0x00000000 被保留以检测空指针异常，从 2GB (0x7FFFFFFF) 到 32 位值的上限 4GB (0xFFFFFFFF) 则保留为操作系统所用。

一旦找到了合适的地址范围以后，OS 就开始查找对应这地址范围的物理内存 (physical RAM) 的操作。内存是分页的，每页在 Win32 下是 4KB。可以通过查找物理内存中未被分配的内存页来确定这些页块；若是找不到合适的空闲物理内存，在 PC 上也可以从其他正在运行的程序处分得一些物理内存。做法是将虚拟内存空间中的一部分内容输出到交换文件中。在物理内存中，或在交换文件中申请分配内存的过程叫做提交内存 (committing memory)。

须知，不是程序能够访问的所有内存地址，都对应经提交的内存。除非操作系统已经被要求向一个地址范围提供内存，所有访问该地址的企图都会被内存控制器捕获并触发一个程序异常。由于内存控制器会负责将物理内存页和虚拟内存页映射到每个程序各自的地址空间里，对某个地址范围对应的存储页面在物理内存中的顺序并没有要求。感谢操作系统，所有程序都能且仅能访问一块连续内存。

1.9.4 重新思考关于数组增大的问题

有几个事件会导致动态数组增大时的性能下降。

首先是当应用程序需要增加一个已有数组的尺寸时，必须分配大于数组现有尺寸的一块新内存空间。操作系统必须在程序的地址空间中找到这个新的内存块，找到空闲的内存来支持它，对内存控制器重新编程来使这些内存页面对程序看起来是连续的。由于必须进行内存复制，现有的一个数组数据的内存块，和较大的一个为增长后的数组准备的较大的内存块，两者必须同时处于活动状态。这意味着，分配内存时可能至少需要两倍于数组数据尺寸的存储空间，根据运行时内存管理器所采用的重分配策略，则可能需要更多。随着动态数组增大，为了为拷贝操作提供空闲内存，而将其他内存交换出内存的概率也随之增大，尽管大约 1/3 到 1/2 的空间会在拷贝完成之后立即被释放。在那些没有交换文件的平台上，比方说 Xbox，即使有足够的空间来容纳新的经扩展的数组，若是空间不足以同时容纳两个缓冲区，也是无法完成数组增大操作的。

其余的性能下降是由于对数组内容进行复制。所有的数组元素被读取，即使没有对它们做任何处理。而 CPU 的高速缓存中一些较重要的数据则可能会因此被擦除。作为复制操作的一个副作用，应用程序的堆也许会因原始缓冲区被删除而变得更充满碎片，由于数组中的数据地址可能会改变，程序的其他部分不能直接使用指向个别数组元素的指针。事实上，程序必须保存元素在数组中的下标，并在每次访问时计算出指针的值。

最终，当动态数组增大的时候，会出现“空间松散”的内存问题。也就是说会出现已经分配但是还没有用来保存任何数据，并因此无法挪作它用的内存。选择较大的尺寸增加值，可以降低数组不得不进行重新分配的次数，但不幸地，同时增大了数组的松散空间的数量。

1.9.5 新的增长规则


本文的关键是要理解程序的地址空间是可以脱离物理内存而进行分配和管理的。既然操

作系统为我们的程序提供了这一功能，我们就能更高效地管理动态数组占用的存储空间。

通过地址空间进行管理（受控）的动态数组与通常的动态数组的主要差异，在于前者在数组初始化之后，就已经分配了一个足以容纳最大个数元素的地址空间范围，然而此时并没有将任何物理内存提交给这个地址范围。然后，随着数组的增长，逐渐有存储页面被提交给新数组元素的地址。这听上去十分简单，但已能带来如下优势。

- 在每次数组增长时，无需对整个数组中的数据进行内存拷贝操作。不用浪费时间拷贝，CPU 的高速缓存中的数据也不会产生反复。
- 显著地减少了必须由操作系统的内存管理模块来进行的作业数量。操作系统只需要找到足够的内存页面来支持数组的扩展部分，而不像老方法那样需要找到能容纳整个新数组的空间。同时亦无须进行新的地址空间管理。
- 由于降低了数组增长时操作系统的负荷，我们可以使用较小的数组长度增量，即较频繁地进行数组增长，以便减少平时占用的松散内存的数量。
- 由于拥有数组副本现在并不意味着必须重新分配缓冲，也由于数组中平时占用的松散内存数量得以减少，虚拟内存系统的负荷也大大减轻，从而减少了交换文件的使用次数。在不支持交换文件的平台（比如 Xbox）上，一些无法通过传统方法找到足够的空闲内存来进行数组增长的情况，现在可能能够增长了。
- 由于数组数据的指针的值不会改变，程序的其他部分可以直接将指向数组元素的指针保存下来，以备不时之需，而无需每次都重新计算指针的值。
- 在数组中的元素被删除或移除后减少数组占用的内存数量成为了一种可行的做法。当数组管理器检测到在已提交的区域尾部存在着空闲的内存页面，就能将其释放。一般说来，释放一个页面的时候操作系统不做任何处理，直到需要满足另一次内存分配需求时才进行处理。

1.9.6 使用地址空间受控的数组

 在本书配套光盘里有 ArrayManager2.h 和 ArrayManager2.cpp 这两个文件，
ON THE CD 其中定义的 CAddressSpaceArrayManager 类提供了一个与 CarrayManager 等价的接口，但其服务的定义采用了上文中描述的地址空间受控的动态数组技术。

在本例中，Initialization()方法有较多的处理。它跟踪数组的最大和当前“使用中”的长度，并均以两种形式记录：一是数组元素的个数，二是内存页面的页数。首先，通过用 MEM_RESERVE 参数调用 VirtualAlloc()来保留能容纳上限个数数组元素的地址空间。然后，用 MEM_COMMIT 参数再次调用 VirtualAlloc()函数，这一次是为了给初始时一定数量的元素提供实际的存储空间。实际提交的数组元素个数保存在 m_Num_Elements_Committable 变量里。注意，我们将其按 Initial_Size 变量的值取整，Initial_Size 变量代表提交的内存页面能容纳多少个数组元素。增加数组元素的时候，检查该值以确定是否需要提交额外的内存页面。除非绝对必要，我们可是不希望提交新页面的。

```
void CAddressSpaceArrayManager::Inititalize
```

```

        (int Maximum_Size, int Initial_Size)
    {
        // 将数组初始化成指定的容量
        // (可将其移到构造函数中去)
        m_Element_Size = sizeof (CArrayItem);

        // 计算我们希望预留的页数
        m_Maximum_Pages_Committable =
            (( m_Element_Size * m_Maximum_Num_Elements )
             + OS_Page_Size - 1) / OS_Page_Size;

        // 备注: 可以用下一个页面的大小
        m_Maximum_Num_Elements = Maximum_Size;

        // 为这一个数组实例保留地址空间
        m_Array_Data = VirtualAlloc(NULL,
            m_Element_Size * m_Maximum_Num_Elements,
            MEM_RESERVE, PAGE_READWRITE);

        if (!m_Array_Data)
            assert("address allocation failed");

        // 根据初始时指定的数组元素个数来提交 (commit) 存储空间

        // 计算有待 commit 的存储量
        m_Num_Pages_Committed =
            ((Initial_Size * m_Element_Size)
             + OS_Page_Size - 1) / OS_Page_Size;

        // 将可 commit 的数组元素数量进行下一个页面的大小
        m_Num_Elements_Committable =
            (m_Num_Pages_Committed * OS_Page_Size) /
            m_Element_Size;

        // 提交内存
        void* result = VirtualAlloc(m_Array_Data,
            m_Num_Pages_Committed * OS_Page_Size,
            MEM_COMMIT, PAGE_READWRITE);

        if (!result)
            assert("memory commit failed");

        m_Num_Elements_Used = 0; // 未曾增加过任何元素
    }

```

在 32 位 Windows 系统下, VirtualAlloc() 这条 API 函数既可以用来保留地址范围, 又可以为预留的地址范围提交内存。页的最小尺寸是 4KB, 反映在 OS_Page_Size 变量里。

AddItem() 函数是类中主要进行处理的部分。它接受一个 CArrayItem 结构类型的指针作为参数, 将它的一个副本添加到数组的末端。它首先要检查数组的长度是否已经达到上限。

若是已经达到上限，则程序员面临的或是产生一个错误，或是退回去采用传统的数组增长方法。成功后，该类第二次检查自身的长度，判断是否需要提交新的内存页面。若需要提交额外的内存，则我们可以控制一次提交多少个新页面。这里我们采用每次增加一个页面的策略，以抑制松散内存。不过你可以在你的实现版本里提供一个调节界面，允许应用程序指定一次提交的内存页面数量。最后我们用 MEM_COMMIT 参数、指定将放置新页面的尺寸和地址信息，调用 VirtualAlloc()来完成额外页面的提交工作。注意，在这里我们将指针转型为字节类型，以避免编译器将地址偏移量放大数组元素大小的倍数。之后内存跟踪变量更新，元素数据被复制到数组尾部。

```
void CAddressSpaceArrayManager::AddItem
    (CArrayItem* the_Element)
{
    if (m_Num_Elements_Used >= m_Maximum_Num_Elements)
    {
        // 此处处理待定
        assert("Array exceeded maximum size");
    }

    // 提交的内存满了吗?
    if (m_Num_Elements_Used==m_Num_Elements_Committable)
    {
        // 向数组的地址空间增加更多的可 commit 的存储空间

        // 有很多种计算方法...
        int Num_Pages_To_Grow = 1;

        void* result = VirtualAlloc(
            (byte*)m_Array_Data +
            m_Num_Pages_Committed * OS_Page_Size,
            Num_Pages_To_Grow * OS_Page_Size,
            MEM_COMMIT, PAGE_READWRITE);

        m_Num_Pages_Committed = m_Num_Pages_Committed +
            Num_Pages_To_Grow;

        // 更新对应着存储的元素个数
        m_Num_Elements_Committable =
            (m_Num_Pages_Committed * OS_Page_Size)
            / m_Element_Size;
    }

    // 向数组中增加元素
    m_Array_Data[m_Num_Elements_Used] = *the_Element;

    // 更新当前使用中的元素个数
    m_Num_Elements_Used++;
}
```

关于选择数组的最大长度有些讲究。应当足够大，几乎不会在实际工作中出现不够大的情况；但又不能过大，以免过度浪费地址空间。在 32 位 Windows 操作系统下，一个程序能访问的总地址空间略小于 2GB。虽然这容量是远大于现在大多数程序所需要的容量，若是胡乱选择数组的最大长度，不久就会落入地址空间耗尽的尴尬境地。

另一项警告就是，在保留地址范围的时候，VirtualAlloc()这个 API 函数会返回在 64KB 边界上开始的内存地址。

RemoveItem()方法很简单，从数组中删除一个元素，并将数组的最后一个元素搬过来填补删除元素而造成的空缺。CAddressSpaceArrayManager 类增加了一项检查，检查是否有内存页面可以释放或解除提交 (decommit)，而将空闲内存归还程序。这是通过 Windows API 函数 VirtualFree()来进行的。在 Windows 上，由于操作系统能够按需结合和切分内存块，并不强求内存必须以提交时的相同大小或顺序来解除提交。一旦认定在数组的尾部有已经提交但没有用到的内存页面，就可以用 MEM_DECOMMIT 参数、需释放内存页面的地址和尺寸调用 VirtualFree()。

```
void CAddressSpaceArrayManager::RemoveItem
    (int the_Element_Index)
{
    if (the_Element_Index < 0 ||
        the_Element_Index >= m_Num_Elements_Used)
    {
        assert("Bad Array Element Index");
    }

    // 移除指定的 item，将数组的最后一个元素挪过来填补空隙
    m_Num_Elements_Used--;
    if (the_Element_Index < m_Num_Elements_Used)
    {
        m_Array_Data[the_Element_Index] =
            m_Array_Data[m_Num_Elements_Used];
    }

    // 判断是否需要释放任何已经提交的页面
    int Unused_Memory = (m_Num_Elements_Committable -
        m_Num_Elements_Used) * OS_Page_Size;
    if (Unused_Memory > OS_Page_Size)
    {
        // 释放最后一页已经提交的内存
        m_Num_Pages_Committed--;
        VirtualFree((byte*)m_Array_Data +
            m_Num_Pages_Committed * OS_Page_Size,
            OS_Page_Size, MEM_DECOMMIT);

        // 重新计算已为之 commit 内存的 item 的数量
        m_Num_Elements_Committable =
            (m_Num_Pages_Committed * OS_Page_Size) /
            m_Element_Size;
    }
}
```

```
}
```

1.9.7 结论

地址空间受控的动态数组通过利用现在操作系统为程序提供的内存管理机制，提高了性能。如果你的数组中的元素个数，在整个容器的生命周期内会有频繁且幅度很大的上下波动，或者某些数组会在某一时刻变得非常大，那么你应当考虑运用控制地址空间的方法。对于小的数据集合，或那些大小保持不变的数组而言，采用上述方法的优势就不甚明显了。对于大的或易变的数据，地址空间受控的数组为传统技术提供了新的手段，并在减少内存消耗的同时增加了提高效率和性能的机会。

1.10 用临界阻尼实现慢入慢出的平滑

作者: Thomas Lowe, Krome Studios

E-mail: tomlowe@kromestudios.com

译者: 沙鹰

审校: 万太平

平滑(smoothing, 也译作校平或修匀)是一个极有用的概念, 对于游戏方方面面的质量都起着重要作用。不使用平滑, 游戏看起来毛毛糙糙、停停动动, 比较生硬; 而使用平滑, 则可以让游戏变得流畅、精致并且更为自然。

本文中使用的“平滑”一词, 指的是一种随时间流逝逐渐调整某个值来逼近目标值的方法。我们可以对任何会随时间改变的值进行平滑, 无论这个值是标量、矢量、颜色还是角度。因此本文描述的方法具有广泛的适用性。下面举几个例子。

- **镜头运动:** 追随某个对象拍摄的镜头的运动常常显得停停动动, 特别是当目标对象本身的运动不规则, 或镜头与全局场景发生碰撞的时候。平滑的镜头运动看起来更为自然, 而且对玩家的眼睛来说也是体贴的。

- **灵活的路径跟随:** 对象可以“平稳”地朝某一点运动。运动参考某条路径, 但并不是刻板地沿这条路径移动。这样就可以避开一些位于路径控制点上的物体。该技术能将路径方向的急剧变化变得平滑, 也能在路径的起点和终点处实现平稳的加速和减速。

- **状态改变:** 实现一个复杂对象的行为时, 尤其是从一个状态进入另一个状态的时候, 位置和速度常常发生不期而至的突发改变。平滑能够削弱这些干扰, 以免分散玩家的注意力。

- **前端:** 将文本或其他对象从屏幕的一处移动到另一处, 改变尺寸或颜色, 这些都可以通过朝期望的值平滑而实现。

本文描述了一种基于临界阻尼弦的慢入慢出的平滑方法。模型实现为一个易用的函数, 提供了一种强大可靠的平滑手段。

1.10.1 可用的技术

鉴于平滑有如此广泛的应用, 值得考虑几种不同的方法。下面我们就概括地比较一下。

1. S 形曲线

如果只是需要从一个静态值经过一段指定时间平滑到另一个静态值,

那么一条 S 形曲线就能解决平滑的慢入慢出运动。从一个正弦波或两个抛物线上取一段弧线，可以实现一阶连续（即速度连续），而二阶连续曲线（即加速度连续）也能够实现，不过稍稍复杂一些（见图 1.10.1）。

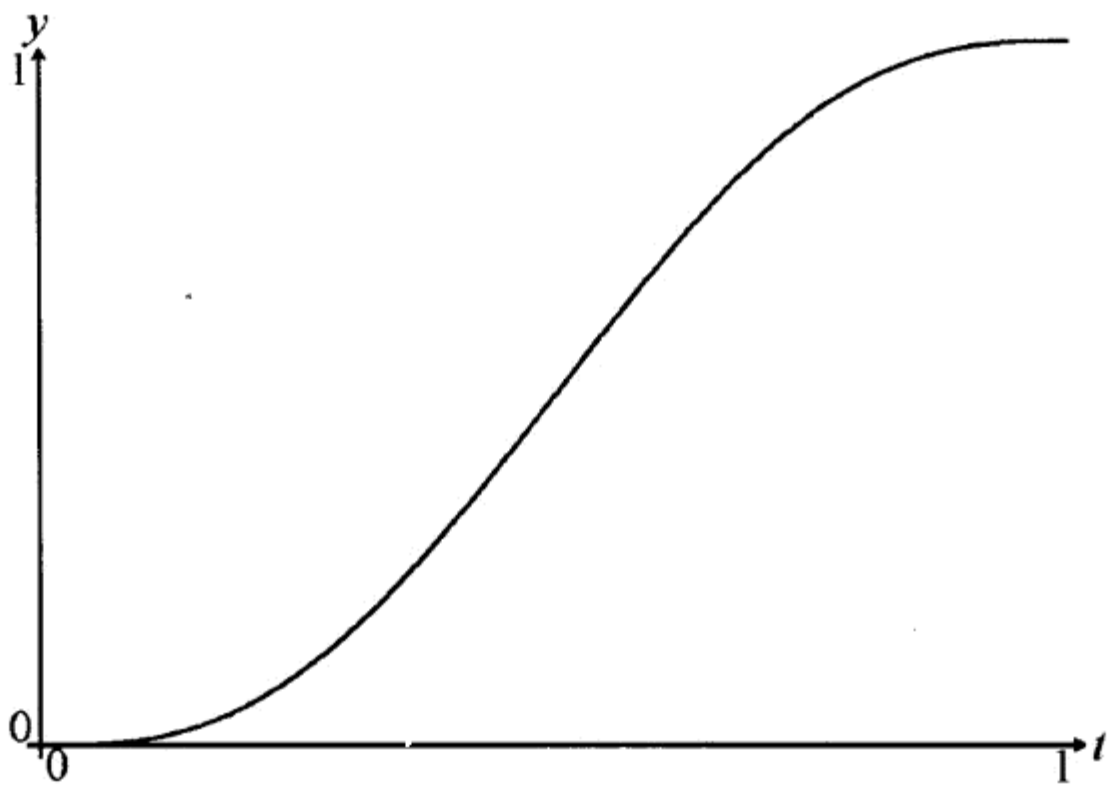


图 1.10.1 二阶连续的 S 形曲线 $y = 6t^5 + 15t^4 - 10t^3$

2. 指数衰减

指数衰减是一个常用的方法，通常形如：

$$y = y + (\text{desiredY} - y) * 0.1f * \text{timeDelta}$$

上式中 timeDelta 是你的代码中两次相邻更新之间的时间间隔，单位是秒。

这一类平滑的优点在于，你可以逼近一个运动中的目标。而且，也不需要保存“已过了多长时间”这样的数据。此方法可被描述为“慢出”，但是请注意，初期的运动是骤变的（见图 1.10.2）。

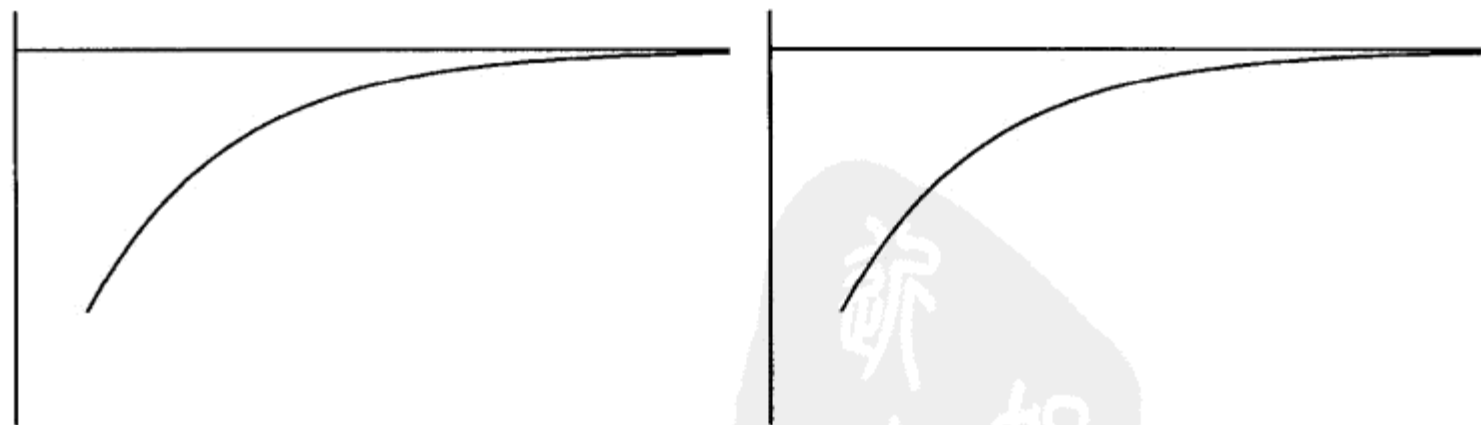


图 1.10.2 指数衰减，左图初始状态为静止，右图初始状态为运动中

3. 临界阻尼弦

本模型将指数衰减的可靠性及 S 曲线的“慢入”属性相结合，能够以连续的速率逼近一

个改变中的目标。这一点不同于必须维护一个速率变量的指数衰减技术。

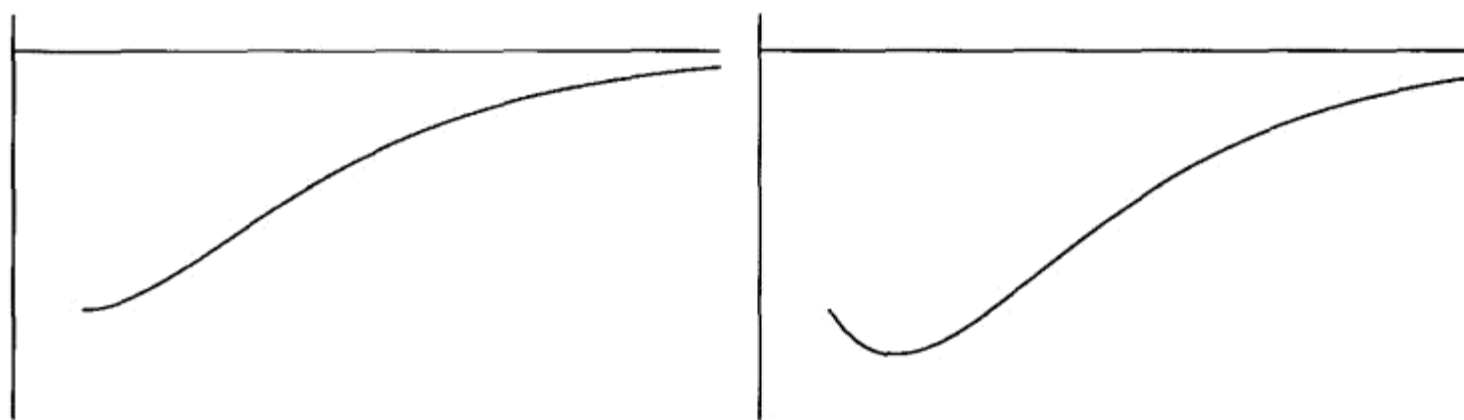


图 1.10.3 运用临界阻尼弦，速度不会发生突跃

1.10.2 阻尼弦与临界阻尼

临界阻尼弦的工作原理是什么？该技术基于阻尼弦（一个有阻力的弹簧）。在弹簧末端的点（ y ）上有一个与弹簧实际长度与其自然长度（即目标位置 yd ）的改变量成正比的力，这是胡克定律（Hooke's Law）。此外，阻尼弦上还有一个方向与 y 点的速度方向相反的阻力。因此，阻尼弦上 y 点的运动规律可以用如下的微分方程来刻画：

$$m \frac{d^2 y}{dt^2} = k(yd - y) - b \frac{dy}{dt} \quad (1.10.1)$$

式中 m 是 y 点的质量， k 是弹簧常数（或称其为弹簧强度）， b 是阻尼常数（或者说阻力的大小程度）。

这些常数影响弹簧恢复 yd 长度的过程，在我们讨论平滑函数这个语境下，此过程即是我们的值接近目标值的过程。给 b 一个较小的值，会造成伸缩过头和振荡；如果给 b 一个较大的值，则有慢慢收敛的效果。临界阻尼在 b 位于两个极值中的情况下发生，既不产生振荡，又能以最理想的收敛速率接近 yd 。当 $b^2 = 4mk$ 时发生临界阻尼。因此，我们可以将式 1.10.1 化简为：

$$\frac{d^2 y}{dt^2} = \omega^2 (yd - y) - 2\omega \frac{dy}{dt}, \quad \text{其中 } \omega = \sqrt{\frac{k}{m}} \quad (1.10.2)$$

ω 是弹簧的固有频率，也可以说是弹簧硬度或强度的度量。

1.10.3 实践

现在看如何实现这个模型。我们的目标是编写一个函数，根据输入的目标位置、时间间隔、以及平滑因子，更新位置和速度，像这样：

```
y = SmoothCD(y, desiredY, velY, smoothness);
```

临界阻尼弦模型（式 1.10.2）可以用标准的数值积分方法来逼近，但是事实上这并没有必要，因为存在一个确切的（分析的）闭合解（参见 [Stone99]）。如式 1.10.3：

$$y(t) = y_d + ((y_0 - y_d) + (\dot{y}_0 + \omega(y_0 - y_d))t)e^{-\omega t}$$

(1.10.3)

上式中 y_0 是初始位置； \dot{y}_0 则是初始梯度，即速率。
用很小的步长对此式进行微分，我们得到：

$$y_1 = y_d + ((y_0 - y_d) + (\dot{y}_0 + \omega(y_0 - y_d))\Delta t)e^{-\omega \Delta t}$$

(1.10.4)

$$\dot{y}_1 = (\dot{y}_0 - (\dot{y}_0 + \omega(y_0 - y_d))\omega \Delta t)e^{-\omega \Delta t}$$

(1.10.5)

两个等式（精确地）给出了经过时间间隔 Δt 后的新位置和新速度，这正是我们需要的。

关于平滑因子，注意我们能够使用 ω 来表示，但是一般来说，用平滑时间而不是弹簧强度来控制平滑函数，那样更为直观。对平滑时间的一种定义是“预期的以最高速度到达目标所需的时间”（见图 1.10.4）。这样的定义是有用的，理由有二。首先，朝移动中的目标进行平滑的时候（由于阻力）它与滞后时间相同，滞后的计算就变得十分简单。其次，它为我们提供了非常简单的换算关系： $\omega = 2 / \text{smooth time}$ ，这是从式 1.10.2 推导而得的。

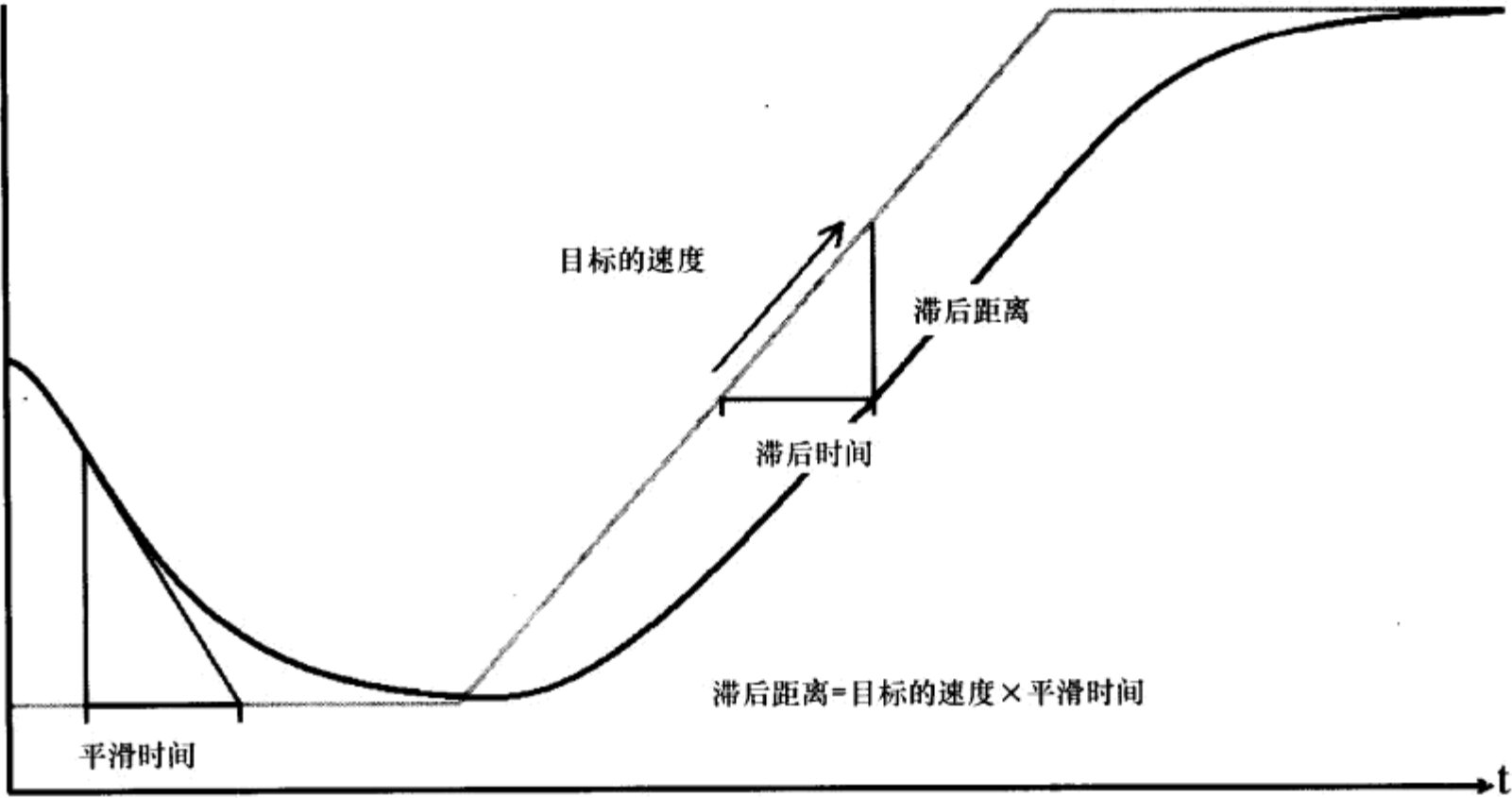
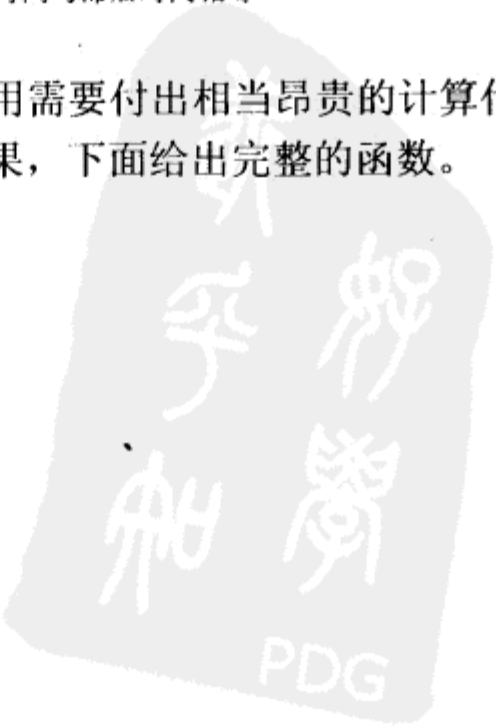


图 1.10.4 平滑时间与滞后时间相等

仍然有一个问题，那就是指数函数的调用需要付出相当昂贵的计算代价。幸运的是，这在我们的使用范围内可以精确估计。作为结果，下面给出完整的函数。

```
float SmoothCD(float from,
               float to,
               float &vel,
               float smoothTime)
{
    float omega = 2.f/smoothTime;
```



```

float x = omega*timeDelta;
float exp = 1.f/(1.f+x+0.48f*x*x+0.235f*x*x*x);
float change = from - to;
float temp = (vel+omega*change)*timeDelta;
vel = (vel - omega*temp)*exp;    // Equation 5
return to + (change+temp)*exp;  // Equation 4
}

```

估计 e^x 的近似值方法是运用如下所示的截断的泰勒展开式。我们这里 $e^{-\omega\Delta t}$ 幂可以作为 $1/e^x$ 来计算，其中 x 为 $\omega\Delta t$ 。

$$e^x \approx \sum_{i=0}^n \frac{x^i}{i!} \quad (1.10.6)$$

可以在经常使用的范围里调节系数，以更好地逼近。对我们的函数来说，这范围基本上就是 $0 < x < 1$ 。采取以上近似方法计算 `exp` 的误差小于 0.1%。在 PC 上，速度也比使用 `exp()` 函数要快上差不多 80 倍！通过使用阶数更高的多项式可以给出更准确的近似值。

1.10.4 设置平滑速率的上限

最后再简短地展示一条便利的扩展：如何来设置平滑速率的上限。由于出发点和运动中的目标之间有滞后距离（大小等于 $s \cdot \text{smoothTime}$ ）的存在，如果和目标之间的距离被限制在不大于滞后距离的范围内，那么 s 就成了速率的上限。

做法是这样的，在值设定之后，再一次改动它，从而较平稳地接近平滑速率的上限。

```

float maxChange = maxSpeed*smoothTime;
change = min(max(-maxChange, change), maxChange);

```

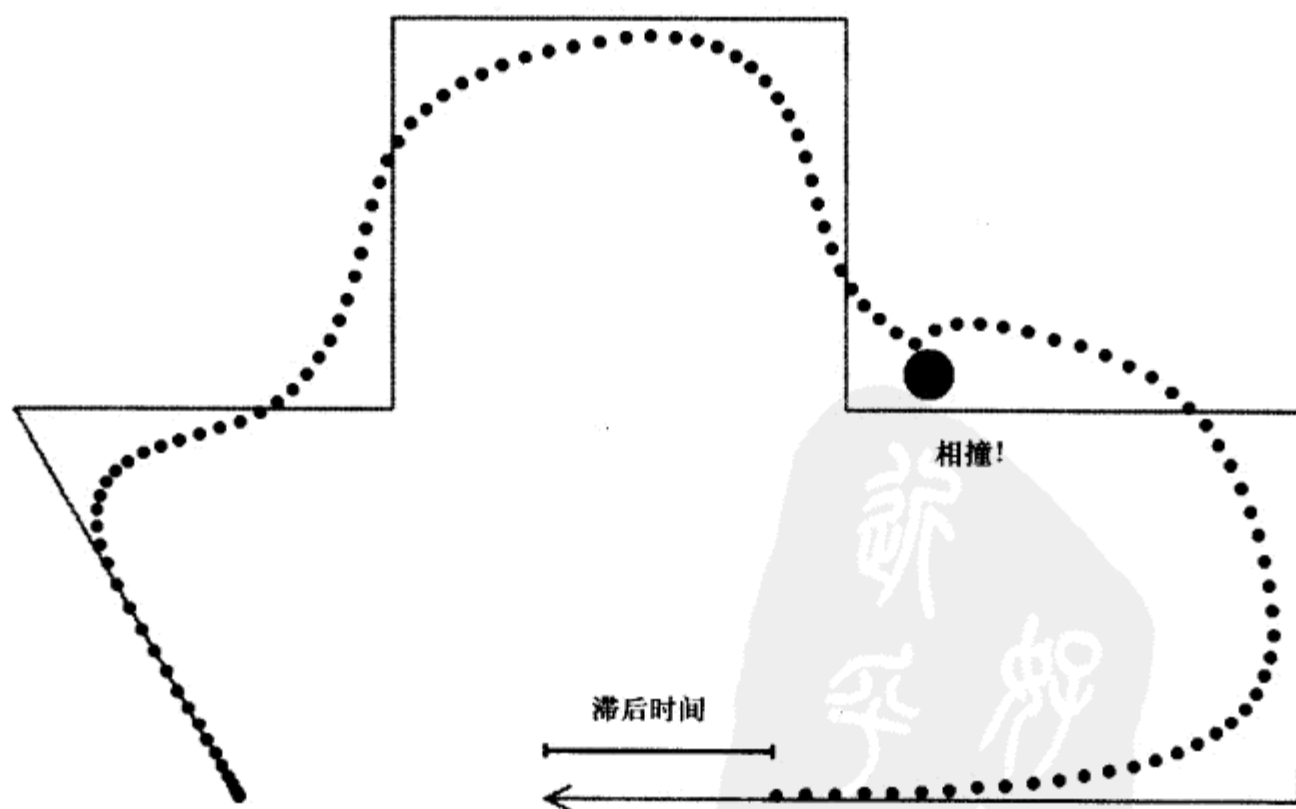


图 1.10.5 矢量平滑，沿路径行走，在途中进行偏转

1.10.5 结论



通过一个简单的算法，就能实现用临界阻尼弦的平滑。在这里给出的实现是对精确解的一个准确逼近，不论时间间隔多大，也不论平滑时间多小，该实现都是稳定的。由于可能需要对实数、颜色、矢量等等进行平滑，采用模版函数将是比较合适的，随书光盘正带有这样一个版本。

1.10.6 参考文献

[Stone99] Stone, B. J., "A Summary of Basic Vibration Theory," available online at www.mech.uwa.edu.au/bjs/Vibration/OneDOF/1DOF.pdf



1.11 一个易用的对象管理器

作者: Natalya Tatarchuk, ATI Research, Inc.

E-mail: natashat35@yahoo.com

译者: 沙鹰

审校: 许竹钧

现代游戏的核心是数据库。数据库负责管理所有运行时的游戏对象，也负责在对象生命周期内提供相关的服务，如创建、解构、缓存、查询、保存、加载等操作。所有这些操作，一般都能在同一个数据管理系统中对不同类型的对象进行。因此，对于开发游戏中数据库组件的程序员来说，关键是灵活性（flexibility）、使用的方便程度（convenience），以及效率（efficiency）。

在以往，我们的解决方案是通过模板，以及其他支撑不同类型对象的灵活创建的相关技术，直接实现对象的分配与管理等操作。在 [Bilas00] 一文中你可以看到如何实现一个基于句柄的模板化的资源管理器；[Hawkins02] 一文则讨论了一种对象管理机制，其中利用了基于句柄实现的灵巧指针。但是，这些方法中的大多数都存在着某种程度的限制，导致游戏开发者们可能会在开发其对象管理系统时落入陷阱。比方说，有时游戏程序员们并不知道在游戏中总共会有多少种对象类型；技术指导有时也会要求必须在项目日程中的任意时刻都可以添加新的对象类型。如果游戏中的对象管理器是封装成一个独立的库而存在，而更多的对象类型会在该库完成之后方被添加，则这些都是相应的问题。若不能灵活地进行对象管理，就很难避免在进行以上操作的同时重新编译整个项目，或至少要重新编译整个对象管理组件。本文针对动态对象管理的问题，讨论了如何在对象创建和删除中使用自定义类，和允许在运行时查询类型信息的自定义的对象管理系统。

1.11.1 对象管理的传统做法

实现对象管理子系统的方法之一，正是大多数编程项目所热衷的常用方法，能无忧无虑地滥用继承。在这种方法的指引下，开发者为对象设计基类，并实现处理基类的对象管理系统。新类通过扩展基类得到子类而产生。对象管理器也能被用来管理所有衍生出的子类类型的对象，只要所有的对象都借助虚函数而具有共通的功能性，同时也正确地实现各自的虚析构函数就可以。该方法有两点主要局限。首先，它要求系统中的所有对象

都必须从同一个起点衍生而成，也就是说所有对象类在编译的时候已经确定，这可是一个不受欢迎的限制，如果开发者决定添加新的对象类，则必须要对基类有所了解，方能支持新类。其次，所有的对象类都必须实现同样的一些底层函数，这在我们将通过同一个对象管理器来管理大量类型迥异，甚至是完全不相干的类型时，显得尤其不方便。

解决对象数据库问题的第二种方法，是使用模板化数据存储以及模板化对象访问函数（accessor function）的对象管理器类。在这种方法里，你所创建的容器类应当是模板化的，可以用于项目中的任意数据类型。该类允许你通过模板成员函数访问和操作存储着的数据。这个方法也有一些问题。只要用过模板，你一定会知道所有模板函数都要先正确地实例化然后才能被使用。这再一次带来了这个问题，所有可能的对象类型组合都要在编译时被开发者所预料到。增添新的类型需要重编译整个系统。

还有一种常用的做法，对象管理器一律通过 void 类型的指针来访问和存储对象，对象的内存管理不在对象管理器中进行。尽管这种方法有些丑陋，但当你了解到它是多么广泛地得到采用时，一定会大吃一惊。由于此方法使对象管理器看起来并不比对象的临时保管处高明多少，乍看起来似乎削弱了对象管理器的功能。

1.11.2 灵活的对象管理器

有了上面的讨论作为基础，下面我们将设计一个在数据驱动的游戏子系统中，极大地允许运行时灵活性（runtime flexibility）的对象管理系统。这一特定实现的关键优势，就在于它能够以一种类型安全的方式进行对象管理，而无需在将游戏系统投入运行之前就洞悉所有的对象类型。也无需在编译时就明确地定义所有的对象类。也可以在运行时没有困难地将新的对象类型加入系统之中。而且，此方法并不强求系统的用户对所有游戏对象使用相同的公共基类。本文描述的系统还通过其自定义的析构机制，支持对象占用内存的正确回收。此方法在实现中用到了模板和多态（polymorphism）。因此文章给出的代码实现需要用支持模板成员的编译器来编译，例如微软 Visual Studio 中的 C++ 编译器。

1. Runtime 类型信息

要使用本文描述的方法，必须要在你的项目中打开动态类型识别（Runtime Type Identification，简称 RTTI）功能。RTTI 是一种支持在程序执行中确定对象类型的机制，在较新的 C++ 编译器中是在语言级别得到支持的。一般而言，运行时的类型信息是在虚函数表中增加了一个额外指针来进行的，因此打开 RTTI 会使程序的大小有微小的增加。RTTI vtable 指针引用了具体数据类型的 `typeinfo` 结构。而对于每个新类，只会创建一个 `typeinfo` 结构的实例。因此，程序大小的增加是很少的。对于大量运用多态的程序来说，RTTI 机制能带来明显的优势。记住，只要 RTTI 打开，编译器就会自动地为你的项目中的所有类生成对应的类型信息。

使用 RTTI 就不得不涉及到的一个问题是执行速度变慢。在 C++ 中，在运行时获得类型信息，主要机制有两种：一是 `typeid` 操作符，二是 `dynamic_cast` 操作符。使用 `typeid` 表达式的作用是很明显的：即通过 vtable 中的指针获得 `typeinfo` 指针。借助后者获得对作为结果的 `typeinfo` 结构的引用。因此，`typeid` 操作符的时间复杂度是常数，其处理也是完全确定性的。

再看 `dynamic_cast` 操作符，首先获得原始对象类型的运行时类型信息，而后从 RTTI 数据中得到转换后的目标数据类型。然后，C++ 的 runtime 系统就目标数据类型是否与原始数据类型兼容做出正确判断。由于在 `dynamic_cast` 的实现内部免不了要对一系列的基类进行检索，使用 `dynamic_cast` 的开销比使用 `typeid` 调用要高。还要注意的，`dynamic_cast` 操作所耗费的时间与牵涉到的类的继承关系链有关，检索复杂的继承结构（尤其是涉及到多重继承的时候）要耗费较多时间。但是，这一项开销的增加通常只是很轻微的，而且能够在运行时动态识别对象类型这一优点，可以说是弥补了开销增加的缺点，只有那些对效率要求特别高的模块例外。关于更深入的讨论 runtime 类型信息用法，请参阅 [Eckel03]。你也可以在 [Wakeling01] 一文中看到如何实现你自己的动态运行时类型识别系统。

有些程序员们认为，既然 RTTI 允许程序从一个匿名的多态指针中取得类型信息，RTTI 就很容易被水平一般的程序员误用。对于许多过去主要搞过程式程序开发的人来说，把程序组织成一重重的 `switch` 语句，并在 `switch` 中应用 RTTI 服务，那会是很诱人的。可是，如果这样做，他们就没有利用多态和封装的优势。这的确是值得关心的，应该尽快为新手上路的 C++ 程序员进行相关培训，教会他们好的面向对象设计建立在良好设计的多态对象的层级结构和模板中，应只在绝对必要时才进行 RTTI 调用。通过本文描述的方法，RTTI 将成为程序员的得力助手，同时并不会导致较差的程序设计风格。

2. 实现对象管理器类



ON THE CD

存储和管理对象的系统实现为 `ObjectManager` 类。随书光盘里提供的 `ObjectManager.h` 包含其具体代码实现。对象管理器类是整个系统的入口。它将所有对象保存在自己的内部对象数据库中，负责管理每个对象的数据，为对象上的所有操作进行类型安全检查，也负责回收对象及其数据。为了简单起见，我们在对象管理器中使用 STL 的 `map` 容器来保存系统中所有对象。在这个实现中，我们将对象的名字当作对象在系统中的惟一标识符。但是，这在（游戏这样的）实时的软件产品中是一种有限制性而且低效的方法。为实际的对象存储和标识符赋值操作，可以容易地设计出较为精致且便于使用的方案。例如，可以对每个对象在创建之时即设定一个 ID，方法是将每个对象在对象管理器中进行注册，从而在接下来的所有操作中可以通过 ID 来识别对象。

在对象管理器的内部，所有对象都保存在对象容器中。容器对保存在其中的对象，提供整个对象生命周期内的内存管理服务。对象管理器支持通过 `AddObjectEntry()` 方法添加新的对象，该方法接受对象的名字作为该对象在数据库中的惟一标识符，而新的对象将存在一个对象容器中。我们通过成员模板来分辨有待加入数据库的特定对象的数据类型。在向数据库中添加对象的过程中，对象管理器创建一个新的对象容器来保存该对象。通过这成员模板，对象容器能在运行时确定对象的数据类型，并且为今后所有对该对象进行的操作提供类型安全检查。我们稍后将讨论对象容器是如何分辨数据类型的。

有二种方法可以从系统中移除对象：`RemoveObjectEntry()` 和 `DeleteObjectEntry()`。前者只是从对象管理器中移除对象，但并不真正删除这个对象；后者则会通过调用对象所属类的析构函数并最终回收该对象占用的内存。对象管理器支持类型安全的数据赋值机制，方法是在

已存在的对象上进行 `SetObjectEntryData` 调用，该调用的参数是对象标识符（在本文中即对象名字）和新对象。考虑到我们是要实现一个真正类型安全的对象管理器系统，新的对象类型必须和已经保存在具有相同标识符的对象容器中的数据类型相匹配，若不匹配则不能进行数据更新。例如，你要往数据库中添加一个名叫“Merlin”的 `SorcererCharacter` 型对象，过后又希望将名为“Merlin”的数据替换为一个 `MedusaCharacter` 类的实例，由于新提供的对象与对象容器中已经存储着的对象的数据类型不一，对象管理器将无法完成赋值。



为了方便起见，对象管理器也提供两种方法来访问存储着的对象的数据类型：`GetObjectEntryType()`方法仅需对象的标识符这惟一一个参数，就可返回以该标识符存储着的数据的实际 `type_info` 指针；`CheckObjectEntryType()`方法通过指定数据和待查的对象标识符来自动检查运行时的数据类型。若你希望支持序列化方法，将对象保存成一个包含许多不同类的实例的数据文件中，该方法会是特别有用的。你也可利用此方法来查找所有属于某个数据类型的对象，将其先行保存。为了追求简单性和可读性，文中（以及配套光盘上的）的类定义，只是基本的实现。但显然这些类可以被扩展以执行更加复杂的对象管理操作。

3. 实现对象容器

`ObjectContainer` 类处在系统的核心。此类是用于在对象管理器中管理具体对象的。因为我们是使用一个含有模板成员函数的具体类（`concrete Class`），而非一个纯粹的模板类，所以在这个实现中，可以添加新的数据类型而无需重新编译管理器类。当我们向对象管理器中增加一个新对象的时候，对象管理器首先创建一个对象容器的实例，以包裹该对象并管理其资源。就本文而言，我们只提供关于如何执行类型检查、数据赋值和替换，以及对象容器中的数据所占内存的回收的例子。下面请看对象容器的具体实现。每个对象容器都保存有下列值：

```
void*      m_pData;
IDestroyer* m_pDestroyer;
type_info* m_pTypeName;
```

在对象容器实例创建完毕之后，所有成员数据被设为空值。但是，对象容器的第一次数据赋值中，`m_pData` 域被设定为指向实际对象的指针（`SetData()`方法的参数之一），参见程序清单 1.11.1。在数据赋值过后，对象容器判断收到的数据的类型信息，为数据的值开辟存储。为了避免可能出现的重名现象，并不单纯地取出数据类型的名称，而是使用与特定类相应的 `type_info` 指针，并且每次进行数据类型匹配时，通过指针获得类型信息。这些处理是在 `SetData()`方法的第 9~13 行进行的。同时也保存新获得的数据的类型标识符（第 16~17 行），为了实现类型安全的数据赋值，这一数据成员的存在是必要的。

程序清单 1.11.1 通过 `SetData` 方法进行对象赋值

```
1. template<class DataType>
2. bool SetData( DataType& const dataValue )
3. {
4.     // 首先确保删除了一切预先保存在
```

```
5.    // 这个物体中的数据
6.    if ( m_pData )
7.        delete_ptr ( m_pDestroyer );
8.
9.    // 为数据的值创建新的存储空间:
10.   DataType* pData = new DataType (dataValue);
11.
12.   // 赋值:
13.   m_pData = pData;
14.
15.   // 保存该对象的类型信息
16.   m_pTypeName = const_cast <type_info *>
17.               (&(typeid(DataType)));
18.
19.   // 为该数据对象中的数据创建
20.   // 一个销毁器
21.   m_pDestroyer = new Destroyer<DataType> ( pData );
22.
23.   return true;
24.)
```

对象容器类通过 `GetTypeID()` 方法为访问存储的对象的类型信息给出了接口, 通过 `CheckType()` 方法使调用者能够验证新输入的对象是否与对象容器实例中已经存储着的对象类型相匹配。 `IsEmpty` 方法可检查这个对象容器实例是空的, 还是其中存有一个对象。

4. 实现 Destroyer 类

在程序清单 1.11.1 中, 我们可以看到容器实例会创建 `Destroyer` 对象。为了确保在删除容器的同时准确无误地回收其中的实例, 这是必要的。让我们仔细看一下内存回收的机制。记住, 我们要实现的目标之一就是要避免每当要支持新的存储类型就不得不重新编译管理器代码。常用的一个在数据库中保存多态对象的方法就是, 采用带有虚析构函数的公共基类, 储存着的对象可在进行 `delete` 操作的时候正确地被删除。只要这个数据库中所有的对象都有一个共同的基类, 这种方法就可以保证所有对象占用的内存都可以被正确地回收。但是, 如果我们希望编写一个不强求所有对象都具有相同基类的系统, 则只能将对象表示为 `void` 指针。但是这一做法也并不是处理对象析构的当然之选。这一设计问题是通过为每个对象容器创建相应的 `Destroyer` 对象来解决的。



ON THE CD

对象容器中保存了指向一个 `Destroyer` 对象的接口指针。程序清单 1.11.2 给出了 `Destroyer` 类的代码实现 (参见光盘上的 `Destroyer.h` 文件)。该接口只负责对象条目的分配与回收, 本身并不是模板化的, 因此我们可以直接在对象容器类中使用它, 而无需在运行时指定模板数据类型。 `Destroyer` 接口的实现思路和常见的、仅在运行时才能确定数据类型的类工厂模式恰好相反。

我们实现 `Destroyer` 机制是分两步进行的。当调用 `ObjectContainer` 类实例的 `SetData()` 方法以便保存实际对象时, 容器会创建一个按照该对象类型而实例化的 `Destroyer` 模板的具体实例。由于 `Destroyer` 实例是根据对象的数据类型来建的模板实例, 它能够正确地回收存储对象

占用的内存。

程序清单 1.11.2 对象管理器所使用的 Destroyer 模式

```
class IDestroyer
{
public:
    IDestroyer () {}
    virtual ~IDestroyer () {}
};

template <class T> class Destroyer : public IDestroyer
{
public:
    Destroyer()
    {
        m_pData = NULL;
    }

    Destroyer ( T* pValue )
    {
        m_pData = pValue;
    }

    virtual ~Destroyer()
    {
        if ( m_pData )
            delete ( m_pData );
    }

private:
    T* m_pData;
};
```

虽然在我们的对象数据库的实现中大量地应用模板，但上文中“对象管理的传统做法”一节中提到的通用的模板化的对象管理，和稍后提到的新方法之间的关键差异，在于后者没有将一个类实现成纯粹的模板类。事实上，我们采用的是具有用来实现对象容器的模板成员函数的实体类。这一点细小的不同，在编译时将对象管理器与其中存储着的数据分离开来。我们因此而可以在项目中的任意时刻轻松地增加新的数据类型。没必要重新编译或修改对象管理器的代码。

1.11.3 结论

在本文中，我们提出了一种灵活的、不需在编译时将类型标识符完全确定的对象管理器实现方法。比起标准的用模板数据存储和单继承方案来说，由于本方法允许在游戏开发周期中的任意一点添加新的游戏类，而无需进行游戏中核心对象管理系统的重新编译，提高是很显著的。

1.11.4 参考文献

[Bilas00] Bilas, Scott, "A Generic Handle-Based Resource Manager," *Game Programming Gems*, Charles River Media, 2000.

[Eckel03] Eckel, Bruce, *Thinking in C++: Practical Programming, Second Edition*, Prentice Hall, 2003.

[Gamma95] Gamma, Erich, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Hawkins02] Hawkins, Brian, "Handle-Based Smart Pointers," *Game Programming Gems 3*, Charles River Media, 2002.

[Wakeling01] Wakeling, Scott, "Dynamic Type Information," *Game Programming Gems 2*, Charles River Media, 2001.



1.12 使用自定义的 RTTI 属性对对象进行流操作及编辑

作者: Frederic My

E-mail: fmy@fairyengine.com

译者: 沙鹰

审校: 许竹钧

载入和保存关卡，并不是一件容易办到的事。而在游戏开发的过程中，用于处理关卡数据的软件工具也在不断地进化，这样一来，载入和保存关卡就更困难了。写作本文的目的是要提出一种方法，尽可能将组成关卡的变量的流操作（streaming）和编辑予以自动化，以便简化甚至彻底消除新旧版本数据文件之间的兼容性问题。文中提出的方法是由三个简单元素共同形成的：扩展的 RTTI（Runtime Type Information）系统，与各类变量关联的属性，还有一个对象工厂（object factory）。

1.12.1 扩展的 RTTI

RTTI 是一个负责保存程序使用到的类的元数据（metadata）的系统。例如，在 [Wakeling01] 一文中描述的实现里，类的 metadata 包含一个 ID（例如类的名称）和一个指向其父类 metadata 的指针（这里不支持多重继承）。可以根据对象的地址来访问这个对象所属的类的 metadata，并在 metadata 指针形成的树结构中找出继承关系。这常用来在运行时检测某个 C++ 对象是否是某个类或子类的实例，以便在使用多态结构的时候能够安全地从基类转型到子类（downcast）。

C++ 语言内建对 RTTI metadata 的支持，每一款较新的编译器都能够为我们生成 RTTI 信息。例如 dynamic_cast 操作符，利用 C++ RTTI 来进行安全地转型。

```
// pBase 指向一个 'Base' 类的对象
// 'Derived' 继承自 'Base'
Derived* pDerived = dynamic_cast<Derived*>(pBase);
```

若 pBase 实际上指向一个子类对象，则 pDerived 包含转型后的对象的地址，否则 pDerived 为 NULL。

如同 [Wakeling01] 和 [Eberly00] 中谈到的那样，我们可以设计自己的 RTTI 系统。自行设计的好处在于不必局限于某个特定编译器的 RTTI 实

现,也不必事无巨细地为每个类都保存 RTTI metadata (因为一些较小的类并不会获益)。而且,通过使用自定义的 RTTI 系统我们能够任意地扩展 metadata,使其包含 C++ 标准的 metadata 中不包含的信息。



ON THE CD

下面的代码(随书光盘中有完整代码)的功能就是刚才所说的这些:在捕捉标准 metadata 之余,我们的 CRTTI 类还具有一个额外的 m_pExtraData 成员变量,借以将任意的应用程序数据和标准 metadata 信息并列地保存(这可能令你想起在整个 MFC 类库中贯彻的类似方法)。

```
class CRTTI
{
public:
    CRTTI(const CStdString& strClassName,const CRTTI*
        pBase,CExtraData* pExtra=NULL) :
        m_strClassName(strClassName),
        m_pBaseRTTI(pBase), m_pExtraData(pExtra) {}
    virtual ~CRTTI() {}

    const CStdString& GetClassName() const
        { return m_strClassName; }
    const CRTTI* GetBaseRTTI() const
        { return m_pBaseRTTI; }
    CExtraData* GetExtraData() const
        { return m_pExtraData; }

protected:
    const CStdString m_strClassName;
    const CRTTI* m_pBaseRTTI;
    CExtraData* m_pExtraData;
};
```

有 4 个宏(macro)可助你将自定义的 RTTI 整合到你的类中,它们是。

- DECLARE_RTTI 在类的定义中增加一个静态 CRTTI 成员,并同时定义用来访问该成员的虚方法 GetRTTI()。
- DECLARE_ROOT_RTTI 用在继承关系树中的根类的定义中。除了添加 DECLARE_RTTI 宏会增加的那些成员以外,此宏还增加 RTTI 系统所需的方法。
- IMPLEMENT_ROOT_RTTI 用在根类的实现文件中,将 DECLARE_ROOT_RTTI 定义的静态 metadata 予以初始化。本宏只有一个参数:类的名字。
- IMPLEMENT_RTTI 与 IMPLEMENT_ROOT_RTTI 很相似,但是用在子类中,有一个额外的参数是父类的名字。[Wakeling01]一文中的实现并不支持多重继承。

下面是实际使用这些 RTTI 宏的例子:

```
// RootClass.h
#include "RTTI.h"
class CRootClass
{
    DECLARE_ROOT_RTTI;
    ...
}
```

```
};

// RootClass.cpp
#include "RootClass.h"
IMPLEMENT_ROOT_RTTI(CRootClass);
...

// Derived.h
#include "RootClass.h"
class CDerived : public CRootClass
{
    DECLARE_RTTI;
    ...
};

// Derived.cpp
#include "Derived.h"
IMPLEMENT_RTTI(CDerived, CRootClass);
...
```



ON THE CD

上面的代码进行了定义及初始化，接下来的代码则展示了一些意义不言自明的宏的使用方法（参见光盘上的 RTTI.h），以及访问标准 RTTI 功能的方法。

```
// 这个宏取代了 C++ 里的 dynamic_cast
Derived* pDerived = DYNAMIC_CAST(Derived, pBase);

// 确认 pBase 的确指向一个 Derived 类的对象实例
if (IS_EXACT_CLASS(Derived, pBase)) { ... }

// 确认 pDerived 指向一个 Base 类或 Base 类子类的对象
if (IS_KIND_OF(Base, pDerived)) { ... }

// 这小段代码取出一个指向 pDerived 所属类 metadata 的指针
// 并在 metadata 树中回溯而得出相关类的层次关系
const CRTTI* pRTTI = pDerived->GetRTTI();
while (pRTTI)
{
    pRTTI = pRTTI->GetBaseRTTI();
}
```

在下一节中，我们将看到怎样利用 `m_pExtraData` 成员变量来增加 metadata 信息，以支持类的属性。

1.12.2 属性

我们可以创建属性（property）来代表类里的变量 [Cafrelli01]。每个属性的组成包括：名字、类型（表明了该变量的内存开销大小）、该变量从类定义的头部开始的偏移量、文字形式的描述（可选），还有一些标志（flag）。通过标志来表示一些信息，诸如该变量是否是可被编辑的（editable）还是只读的（read-only），是否需要被保存，等等。一定要注意，每个属

性只能在特定类中被定义一次，然后即被该类的所有实例所使用。这就解释了它为什么不包含指向指定变量的指针，但却包含一个偏移量（与对象实例的地址相加，以访问变量的值）。

在已有的类中开始定义属性之前，我们必须通知 framework 我们希望将属性保存在类的 metadata 中。对指定对象而言，这允许我们访问对象所属的类（以及继承而自的基类）的属性。这正是我们对对象实例进行编辑和 streaming 操作时所需要的功能。

为简化该操作，在 ExtraProp.h 中定义了两个宏：DECLARE_PROPERTIES 和 IMPLEMENT_PROPERTIES。前者的使用方法如下：

```
class CMyClass : public CPersistent
{
    DECLARE_RTTI;
    DECLARE_PROPERTIES(CMyClass, CExtraProp);
public:
    // ... 寻常接口 ...
protected:
    bool m_boSelected; // 例变量
};
```

每个要使用 RTTI 编辑/保存系统的类，都必须继承 CPersistent 类。稍后我们会看到这个类是负责 streaming 处理的。当然，如果一个类需要 RTTI 支持，但并不希望定义任何属性，那么在定义时可以仅使用 DECLARE_RTTI 宏。

DECLARE_PROPERTIES 宏有两个参数：CMyClass 是包含该宏的类的名字，CExtraProp 是另一个类的名字。后者是从 RTTI 中的 CExtraData 类继承来的，负责保存额外的 metadata，其中保存着一个属性列表。DECLARE_PROPERTIES 在 CMyClass 中声明了一个静态成员：CExtraProp。同时也插入了一个用来访问它的静态函数 GetPropList()。这个宏最后还声明了一个静态方法 DefineProperties()，你可以在 cpp 文件中找到它的实现。

```
#include "MyClass.h"
#include "Properties.h"

IMPLEMENT_RTTI_PROP(CMyClass, CPersistent);
IMPLEMENT_PROPERTIES(CMyClass, CExtraProp);

bool CMyClass::DefineProperties()
{
    // 静态
    REGISTER_PROP(Bool, CMyClass,
                  m_boSelected, "Selected",
                  CProperty::EXPOSE|CProperty::STREAM,
                  "help or comment");
    return true;
}
```

IMPLEMENT_RTTI_PROP 是 IMPLEMENT_RTTI 的一个新版本（这两个宏是互斥的），它对类的 RTTI 数据成员进行初始化，使其指向由 DECLARE_PROPERTIES 宏定义的 CExtraProp 对象实例。还有一个 IMPLEMENT_ROOT_RTTI_PROP 宏，当需要在根类中支持属性的时候，可替代 IMPLEMENT_ROOT_RTTI 宏。在这些宏的帮助下，我们在我们的 RTTI 系统和类中的属性之间建立了联系。

IMPLEMENT_PROPERTIES 和 DECLARE_PROPERTIES 接受同样的参数。它负责实现静态的 CExtraProp 成员，并将 DefineProperties() 这个函数指针作为参数传给 CExtraProp 的构造函数。

正如它的名字表示的那样，DefineProperties()的功能是初始化其所属类的属性。当构造用来保存属性的 CExtraProp 类的实例的时候，初始化将自动地进行一次。

最后，REGISTER_PROP 为类添加新的属性。比如说有一个名叫“Selected”的布尔型属性（Bool），与类 CMyClass 中的 m_boSelected 变量有联系。该属性是可被编辑的（CProperty::EXPOSE 标志），备注栏写着“帮助或注释”，并允许被流式地输出到外部文件中（CProperty::STREAM 标志）以备后用。就像这里用的 Bool 一样，属性的各种类型是通过一个定义在 Properties.h 中的一个枚举值来定义的。

这个例子是故意写的这么直接易懂的：我们马上将会看到，DefineProperties()能包含宏列表以外的元素。表 1.12.1 给出了范例程序中实现了的属性。

表 1.12.1 范例程序中实现了的属性

值的类型	在 REGISTER_PROP 时使用的名字	属性类
bool	Bool	CpropBool
float	Float	CpropFloat
unsigned long	U32	CpropU32
字符串 (CString)	String	CpropString
2d / 3d / 4d 向量	Vect2D / 3D / 4D	CpropVect2D/3D/4D
指针、智能指针	Ptr, SP	CpropPtr, CpropSP
其他	Fct	CpropFct

为已有的类增加属性的步骤总结如下。

- (1) 若该类尚未支持我们的 RTTI 系统，先增加 RTTI 支持。
- (2) 将 CExtraProp 类作为第二个参数，调用 DECLARE_PROPERTIES 和 IMPLEMENT_PROPERTIES 宏。这将在类的 metadata 信息块中增加一个属性列表。
- (3) 通过将 IMPLEMENT_RTTI 替换为 IMPLEMENT_RTTI_PROP（或替换为其用于基类的对应宏），在 RTTI 系统和属性之间建立联系。
- (4) 实现 DefineProperties，调用 REGISTER_PROP 来创建自己的属性定义，从而将属性与用于编辑或流操作的变量联系起来。

1.12.3 编辑属性

为类定义好属性后，下一步就是要利用属性来显示和修改内存中的对象实例的内容。

1. 显示值

为了显示对象实例中变量的值，我们需要访问类中的 metadata，取得保存在 metadata 中的属性，要求属性返回该对象实例中某个变量的值。请看如下代码：

```
// pObj 指向一个由 Cpersistent 派生而来的类的对象实例
const CRTTI* pRTTI = pObj->GetRTTI();
while (pRTTI)
{
    CExtraData* pData = pRTTI->GetExtraData();
    CExtraProp* pExtra =
```

```

        DYNAMIC_CAST(CExtraProp,pData);
    if (pExtra)
    {
        CPropList* pList = pExtra->GetPropList();
        if (pList)
        {
            CProperty* pProp = pList->GetFirstProp();
            while (pProp)
            {
                // 进行任何处理, 例如:
                Display(pProp->GetValue(pObj));
                // ...
                pProp = pList->GetNextProp();
            }
        }
    }
    pRTTI = pRTTI->GetBaseRTTI();
}

```

在上面一段代码中, 可以看到属性有一个虚方法 `GetValue()`, 它接受对象的地址作为参数, 以字符串的形式返回相应变量的值。这恰好是我们在控制台窗口或编辑控件等处, 将值显示出来所必需的。不过, 也可以按确切类型来返回值, 请看下面这一段代码:

```

// pProp 是一个 CProperty* 类型的变量
CPropFloat* pFloat = DYNAMIC_CAST(CPropFloat,pProp);
if (pFloat)
{
    float fValue = pFloat->Get(pObj);
    ...
}

```

区别在于, 此处我们必须在执行恰当的转型之前, 事先知道属性的确切类型。如上所示, 因为属性类使用我们自定义的 RTTI 系统, 类型验证是轻而易举的。



CProperty 中还有另外一些方法, 用于获取各种其他在属性注册的同时定义的数据 (如名字、类型、帮助文字、标志)。在配套光盘中的例程序里, 这些方法被用来朝一个 MFC 表格控件 (grid control) 中填充 [Maunder02] 属性数据。利用表格控件来表示类, 这避免了编写大量自定义的对话框的麻烦, 也就减轻了代码维护的负担, 对于用户而言有一个统一的界面更是好事。

2. 修改值

大多数情况下, 对与属性相关联的值进行修改和将其显示出来一样简单。

- 若新的值是从控制台窗口或编辑控件中传来的文本, 首先要将这文本传给 **CProperty** 类的 `SetValue()` 方法: 若文本与属性的类型并不相符 (比如属性是浮点数类型, 却读入了一个 "a00"), 属性相应的变量的值维持不变, 并且 `SetValue()` 返回 `false` 报错。若类型相符, 则值被转换, 变量被修改。

- 若是知道属性的真正类型，我们可以对其进行类型转换，通过该类的访问操作符（accessor）来设定新值。

不过，有一些属性需要特殊的编辑方法，常见的例子如指向其他 `persistable` 对象实例的指针。特殊的编辑方式主要有两类。

- 我们不希望将地址作为 16 进制数显示出来，因为用户无法理解一个地址值。实际上，我们希望将被引用的对象的逻辑名显示出来。

- 用户能够在一个列表选取要引用的对象，该列表列出了相应类型的现有对象。

其他类型的属性也能够从特殊编辑方式中获益，例如在调色板中选取颜色，或以欧拉角度的形式输入四元数（quaternion）。为了处理这些需求，范例程序中的 `framework` 调用了下面几个 `CPersistent` 类的虚方法，从而绕过了默认的显示和修改行为：

- `SpecialGetValue()` 负责提供要显示的文本。例如，对于指针属性，我们可以返回被引用对象的名字，而不直接返回地址。

- 当需要支持特殊编辑方法时，`SpecialEditing()` 负责处理用户的输入。这通常分两步：打开相应的对话框，处理结果。例如可以用它来支持用户在列表选取对象。

- 每当用户输入了新的值时，`ModifyProp()` 在 `SetValue()` 之前被调用。在直接调用 `SetValue()` 不敷使用的情况下，`ModifyProp()` 允许程序员对属性输入执行一些额外的处理。



实现细节请参见配套光盘中的代码。

1.12.4 保存



配套光盘上的例程序支持将对象保存为 XML 格式，也可保存为二进制格式。在本文中，我们主要关注可读性较高的、由 tag 隔开数据块而成的——XML 格式。

每个对象数据的定义都由 `<data class=... ID=...>` 这个 tag 开始，由 `</data>` 这个 tag 结束。`class` 参数用于识别对象的类型，这在载入对象需要重新创建这个实例时有用。`ID` 代表该对象实例，作为能被其他对象所引用的名字。显然这些 `ID` 必须是惟一的，那么该如何生成它们呢？在我们的这个实现中，我们采用对象在内存中的地址作为 `ID` [Eberly00]。这一做法的主要缺陷在于，若我们将某个关卡重复载入和保存多次，就算期间并未进行任何修改，由于对象实例的地址在任意两次程序运行中都可能有所不同，得出的关卡文件也会不同。

是 `CPersistent` 类提供了保存对象并将其所有属性写入磁盘的 `service`。每个可以进行流操作的（`Streamable`）对象都继承自 `CPersistent` 类。这一过程与之前我们看到的显示实例的变量值的操作非常相似。但是在这里，有关指针的问题需要特别处理。

当保存一个对象，而这个对象包含一个引用着另一个 `persistable` 对象的指针属性时，要执行下列步骤。

- （1）和任何其他属性一样，该属性将值写入文件。对于指针属性的情况，值是指向对象的地址。像前面说过的那样，内存地址直接用作文件中的对象 `ID`，因此无需转换。

(2) 如果指针不是 NULL, 它所指向的对象地址被添加到一个在系统内部自动维护的引用列表中。

(3) 当处理完当前对象的流操作后, 依次从引用列表中取出所有地址, 并将其指向的对象进行保存。

(4) 由另一个类记录已经保存过的对象的地址, 以避免在同一个文件中对单个对象实例进行多于一次的初始化, 也提供了对循环引用的数据的支持。

这就是为什么只要保存场景的根 (scene root), 整个场景就会递归地被保存下来以备后用。

1.12.5 载入

将保存下来的数据文件重新载入的时候, 必须要根据从文件中取出的类的 ID 来创建对象。对象工厂 [Alexandrescu01] 正是为这个问题而设计的。创建对象实例时, 必须读取其中包含的数据。遍历保存下来的属性, 读入每一个属性的值, 并将其赋予相应的变量, 就像这个值是用用户手工输入的一样。

对于指针的情况, 仍然有问题有待我们解决。问题是, 不但新的对象实例很有可能与它们先前被保存时具有不同的地址, 而且我们希望指向的对象也许还没有创建完成。因此, 我们需要在保存下来的实例 ID 和实际对象的地址之间建立某种映射关系, 也就是在创建对象之后还要执行一个步骤: 链接。

链接 (linking)

链接意味着, 当所有对象载入完成后, 要将指针属性的值全部替换为被引用实例的实际内存地址。为此, 我们可以创建一个 STL map: 键值 (collection key) 是读取得到的对象 ID, 相应的值 (associated value) 是由类工厂返回的新地址。当加载操作从文件中读取一个指针属性的值时, 要执行以下步骤。

(1) 属性将其指针设为 NULL。这是为了确保每条指针都被初始化为一个可以测试的值。

(2) 对象的 ID 就是该对象被保存 (persist) 时的内存地址。因此若属性载入的 ID 等于零, 我们可以推断这保存下来的对象在得到保存的时候就具有一个 NULL 值指针了。由于属性已将指针设为 NULL, 下面的步骤就没有执行的必要, 不会被执行的。

(3) 若 ID 不等于零, 则属性创建一个 CLinkLoad 对象并将其添加到一个列表中, 该列表包含当载入完成时需要恢复的全部链接。在 CLinkLoad 实例中保存着拥有指针属性的对象的地址, 属性的地址, 和对象 ID。

当所有对象都被创建和加载后, 对链接进行处理, 过程如下。

(1) 对列表中的每一个 CLinkLoad 实例, 在已经创建的对象组成的 map 中查找被引用的对象的 ID, 并取得相应的内存地址。

(2) 用该地址作为参数, 调用 CLinkLoad 对象中引用的属性的 Link() 方法。

这条方法的功能是将地址赋予相关的指针变量。

如果指针的链接失败, 比如在 map 中没有找到相同的 ID, 该对象也不会包含无效的指

针值，而只会在加载时通过属性得到 NULL 值。这个方法可能并不完美，但它确实避免了创建使用后未清除的垃圾指针（dangling pointer）。一般而言，链接不可能失败，但若确实失败了，可能意味着该文件已被损坏或破坏过。

最后，通过遍历链接操作中用到的 map，对每个加载的对象调用 CPersistent 类中的 PostRead() 方法。因为该 map 中包含所有由对象工厂返回的对象，因此这就使得每个类都执行各自特定的初始化操作 [Brownlow02]。

1.12.6 与旧版本文件的兼容性问题：类的描述

有了流操作系统，下面让我们来看一下，当有人修改了（比方说增加了新的）类的属性时会发生什么情况。由于在程序中注册在案的属性与之前用旧版本程序保存下来的属性不再对应，因此程序无法再从这个文件中读取数据。

我们可以这样：存在数据文件中也好，存成单独的文件也好，总之将其中包含的 metadata 数据的描述也保存下来。也就是说，在每个类写入到文件中时，将该类的所有属性（名字和类型）列表予以保存，同时也将该类基类的相关数据予以保存。例如，某个游戏引擎可以将一个球对象的实例保存为下面形式的定义：

```
<class name="CRefCount" base="">
</class>

<class name="CPersistent" base="CRefCount">
  <prop name="Name" type="String"/>
</class>

<class name="CEngineObj" base="CPersistent">
</class>

<class name="CEngineNode" base="CEngineObj">
  <prop name="Subnodes" type="Fct"/>
  <prop name="Rotation" type="Vect4D"/>
  <prop name="Position" type="Vect3D"/>
  <prop name="Draw Node" type="Bool"/>
  <prop name="Collide" type="Bool"/>
</class>

<class name="CEngineSphere" base="CEngineNode">
  <prop name="Radius" type="Float"/>
  <prop name="Section Pts" type="U32"/>
  <prop name="Material" type="Fct"/>
</class>
<data class="CEngineSphere" id="0xD7E7C0">
  sphere0001
  0
  0; 0; 0; 1
  10; -0.5; 0
  true
```

```
    true
    1
    8
    0x0
</data>
```

可见, CPersistent 类是从另一个叫做 CRefCount 类(这是一个引用计数类, 参见 [Meyers96]) 继承而来的。CEngineObj 类的描述中并不包含任何属性定义, 但这并不表示该类没有成员变量, 只是说它没有需要被序列化保存的成员变量而已。

在刚才的例子中, 得到保存的对象地址是 0xD7E7C0, 其名字是 “sphere0001”, 球的位置是 (10; -0.5; 0), 等等。如果另一个 CEngineSphere 类(或其他父类如 CEngineNode) 的实例也被保存在同一个文件中, 那么类的定义并不会被重复保存, 只是会写入新的 <data ...> 数据块而已。这里我们用到了几种类型的属性, 有布尔型、浮点数、32 位整数、字符串、矢量等。稍后我们会讨论有关 “函数” 类型 (“Fct”) 的特殊情况。

1.12.7 与旧版本文件的兼容性问题: 匹配

假设我们的类的描述已经被存进一个文件, 类的实例被存进另一个单独的文件。我们的载入子程序首先取出描述(可能已经过时了), 将其与内存中当前版本软件中的描述进行比较。这一步称为 “匹配”, 试图在文件中的类的属性和内存中的类的属性之间建立联系。具体情况有三种。

- 某个类在可执行文件中的属性和它外部文件中的属性具有相同的名字和类型。在这种情况下, 属性将获得外部文件中的数据。你能看到, 映射并不是按照属性在描述中出现的顺序建立的, 而是需要比较属性的名字和类型。这使得我们可以改变次序、交换、删除或插入属性, 甚至可以将属性移到有继承关系的另一个类中去(在前面的例子中, 我们可以决定说: Collide 标志应该是 CEngineObj 类的成员, 而不是 CEngineNode 的成员, 在这样做的时候我们依然可以载入之前保存而成的文件)。此系统的限制也是很显然的: 在类或其父类的属性列表中, 不可以同时存在两个或以上的对象具有相同的类型和名字。为了执行该规则, 可以在注册属性的时候进行测试。

- 在可执行文件中并没有找到与文件中的属性相同的属性。在这种情况下, 属性是不被使用的 (obsolete), 其值被忽略。更确切地讲, 一条叫做 ReadUnmatched() 的虚方法将被调用, 因此应用程序可以提供一些自定义的行为(例如在日志文件中输出一条警告)。

- 有一个在可执行文件中出现的属性, 在文件中找不到它。在这种情况下, 属性不会收到任何数据, 相应的变量将维持由类的构造函数赋予的默认值。每当在文件已经保存之后新建属性, 就会出现这种情况。再次保存该文件, 则新的属性同时会被添加到描述和数据中。

执行属性匹配的代码主要分布在 CPersistent 类的 RecursiveMatch() 和 MatchProperty() 方法中。该类在文件中的每个属性与可执行文件中的属性之间建立联系(如果存在联系的话)。当载入文件时, 对象数据从文件里被载入可执行文件中的相应属性。因此, 不论有多少对象要写入文件, 匹配对于每个类总是只执行一次。

1.12.8 “函数”属性

到目前为止，我们的实现已能够处理一些简单类型（布尔型、无符号长整数、浮点数）、类（字符串、矢量）以及指针。每个属性对应类中的一个成员变量，因此属性的大小是已知的。可是如果我们要保存容器的内容——比方说指针列表呢？此时，我们遇到了一些根本问题。我们既无法事先知道该容器中有多少个对象，不同的容器中对象的类型、访问方式也都不同。这些特殊情况都由 CPropFct 类管理。

“函数”类型的属性让我们指定在 framework 执行 Get（从属性变量转换到字符串）、Set（从字符串转换到属性变量）、Write（写入文件）、Read（从文件读入）或 Link 操作时调用的函数的地址。下面是从 CEngineNode::DefineProperties() 中提取出来的一段代码：

```
CProperty* pProp = REGISTER_PROP(Fct,...);  
CPropFct* pFn = DYNAMIC_CAST(CPropFct,pProp);  
pFn->SetFct(NULL,NULL,WriteNodes,ReadNodes,LinkNodes);
```

其中有些指针可以是 NULL。在前面的例子里，“Subnodes”属性只支持 streaming 和 linking 操作。作为参数的这三个函数指针的功能分别为处理保存、载入和链接一个元素为节点指针的 STL 表。

- WriteNodes() 首先写入容器中保存的指针数量，然后依次写入各个指针的值 [Beardsley02]。

- ReadNodes() 首先读取保存着的指针个数，然后为其中的每一个都创建一个 CLinkLoad 对象，在链接阶段会用到的。

- 在每个由 ReadNodes() 读入并创建的对象上调用 LinkNodes()。它将被引用的地址插入已加载的对象的节点列表中。

当然，这只是个例子。一个类可以根据需要，支持任意数量的“函数”属性。

1.12.9 技巧和提示

在我们现有的属性和 RTTI 系统中，有下面一些技巧可用。

- 可以将数个属性映射到同一个变量上。例如，你可以定义一个属性将角度按弧度单位（这是游戏引擎能够直接进行计算的单位）来保存，再定义另一个属性将角度按度（degree）来保存。

- 类中的属性保存在类的 RTTI 的附加数据（即 CExtraProp 类）中。当然我们也可以通过继承 CExtraProp 类，为类添加其他数据，而不会影响之前描述过的机制。请注意，仅当这新增数据需要和继承关系配合使用的时候，才有必要往 CExtraProp 的子类中添加数据，否则只需要将这数据记为静态变量就足够了。

- 在注册属性的时候，要指明该属性是否将在用户界面中被显示出来，以及是否是只读的。但是有时候，你可能会希望在一个类的实例中显示某个值，但在该类的所有子类中隐藏该值。更进一步，你可能会希望某个属性仅对特定的对象实例而言才是可以编辑的。例如，在编辑工具中，一些 camera 的位置是固定的（座标属性是只读的），但另外一些 camera 可以

自由运动。你可以通过在牵涉到的类中重载 `IsPropExposed()` 和 `IsPropReadOnly()` 这两条方法来实现该功能。

1.12.10 思考

若能实现下面这些功能，将使我们的 RTTI 系统更为有用。

- 文件兼容性问题仅在游戏开发的过程中需要解决。当游戏软件开发完毕后，所有的文件都（应当）保存为各自的“最终版本”。因此，可以删除属性的文字描述，若载入子程序没有找到这些描述，必须充分信任应用程序，并按照可执行文件中定义的格式读取数据。

- 这个系统并不支持集合体（aggregation），也就是一种由其他类充当成员变量的类。当一个类的实例被保存，该实例中的所有属性都被写入文件。使用一种新的属性类型应该就足以为系统增加该功能了。不过到目前为止，这功能并不必要，因为可以利用指向对象实例的指针属性作为通融办法。

- 当属性的值发生了变化（通过 `ModifyProp()` 或 `SetValue()`）时，变化操作可以被一个 singleton 管理器侦测到。在体系上作此变动后，在大多数情况下，可以很直观地实现 UNDO（撤销值的变化）功能。

1.12.11 结论

本文介绍了这样一种方法，通过向每个类中维护的自定义的运行时类型信息（RTTI）中添加特定的属性，而使对 C++ 对象的编辑、载入和保存操作自动化。对象之间的链接（例如指针）也被考虑在内，且能够在载入时得以重建。在保存属性的同时也保存属性的描述，从而我们可将其与编译在当前执行文件中的 metadata 进行比较，干净利落地处理潜在的新旧版本文件不兼容的问题。

使用本方法的代价并不十分高昂：属性是静态对象，不会占用很多内存。耗时最多的操作就是在载入文件时，将在保存得到的文件中读出的属性与可执行文件中的属性定义进行比较的操作，而这操作对每个类只执行一次。在光盘上的例子程中实现的二进制格式消除了许多字符串转换操作，而这些字符串转换在生成直观可读的 XML 格式时是必需的。

1.12.12 参考文献

[Alexandrescu01] Alexandrescu, Andrei, *Modern C++ Design*, Addison-Wesley, 2001.

[Beardsley02] Beardsley, Jason, "Template-Based Object Serialization," *Game Programming Gems 3*, Charles River Media, 2002.

[Brownlow02] Brownlow, Martin, "Save Me Now!" *Game Programming Gems 3*, Charles River Media, 2002.

[Cafrelli01] Cafrelli, Charles, "A Property Class for Generic C++ Member Access," *Game Programming Gems 2*, Charles River Media, 2001.

[Eberly00] Eberly, David H., *3D Game Engine Design*, Morgan Kauffman, 2000.

[Maunder02] Maunder, Chris, "MFC Grid Control 2.24," available online at www.codeproject.com/miscctrl/gridctrl.asp, July 14, 2002.

[Meyers96] Meyers, Scott, *More Effective C++*, Addison-Wesley, 1996.

[Wakeling01] Wakeling, Scott, "Dynamic Type Information," *Game Programming Gems 2*, Charles River Media, 2001.



1.13 使用 XML 而不牺牲速度

作者: Mark T. Price, Sudden Presence/phobia lab

E-mail: mark@suddenpresence.com

译者: 万太平

审校: 肖丹

制作新游戏最大的挑战之一, 就是如何生成游戏中那海量的数据。像 XML 这样的标准数据元格式 (data meta-format), 能够通过重用已有的工具, 方便数据的创建和编辑。然而, XML 也有一些缺点, 最大的问题就是数据过大、载入和解析 (parse) 过慢。

在本文中, 我们介绍一种新的二进制流式文件格式 XDS (即 eXtensible Data Stream), 在大致达到 XML 的表达力的同时, 避免了 XML 的这些缺点。我们也给出既支持 XML 又支持 XDS 数据的工具集 (toolkit), 你可在游戏制作周期中在 XML 和 XDS 两者之间安全地切换。

1.13.1 为什么要使用 XML 呢?

XML 在几年之前 [W3C100], 就已经被证明其在开发性和协同方面的价值, 而当制作一个新游戏的时候, 这两件事情通常没有在优先考虑之列。在那同时, 环绕在 XML 周围的大肆宣传从来没减弱过。好像在你所能到达的地方, 就有人在吹捧 XML 是以前不知道的解决问题的方法。那么, 这些吹捧真的言过其实吗? 难道就真的和游戏一点边都不沾?

美好的事物

XML 并不是无关紧要的。在现代软件开发中它是一个很重要的工具, 使用它所带来的好处远远超过其缺点。虽然它不是万能的, 但它必定会在你的开发过程中占有一席之地。

虽然有很多可选的存储技术 (见 [Olsen00] 和 [Boer01]), 然而 XML 超过其他任何特有的数据格式, 其专有的最大的优点是它有一个制定得很好的标准且有一个相对庞大的产业作为后盾。确实有成百上千个现成可用的 (off-the-shelf) 工具, 你可以立即拿来为你的项目服务,

从编辑器 (editor) 到展示工具 (presentation Tool) 以及转换工具。通过使用 XML 用于游戏数据, 你可以最大程度地利用这些工具。例如, 你采用一个 XML 编辑器和增加一个 XML schema [W3C301] 来描述你的游戏的数据时, 无需写一行代码, 你就能得到一个简单层次的编辑器用于你的游戏。这意味你的设计团队能够很快开始产生游戏数据。

假如你的游戏数据在开发中需要改变, 你将无需删掉所有已经生成的数据, 也不需要写一个一次性的程序将其转换为新的数据格式。事实上, 你能够简单地编写一个 XSL 转换 (XSL transformation), 用它将数据转换成新的格式 [W3C299]。

使用 XML 允许设计人员在游戏中改变且立即看到它们。相反, 假如他们看到在游戏中有些奇怪问题, 他们能够打开 XML 文件然后使用编辑器或浏览器来检查、识别和改正问题。

天堂中的陷阱

然而, 在 XML 世界中也不都那么完美。从一开始, 读取 XML 比读取一般应用程序中特定的二进制文件格式更加复杂。因此普通的 XML 库庞大而复杂。对于一完整特征的 XML 库来说, 超过 1M 的编译代码量不算奇怪。试图把那个压进你那业已拥挤不堪的 (memory footprint) 存储区域。XML 的负担也扩展到数据文件占据空间数和所花费读它们的时间总量。

最后, 在允许设计者从浏览器中检查的同时也允许终端用户也做同样的事情。虽然这对于培育游戏的 MOD 社区有很大的帮助, 但同时也泄露了游戏中所有你不想让玩家知道的秘密。

很明显 XML 的大部分优点都来自于设计和开发的阶段, 而大部分二进制数据的优点则来自发布阶段。如果我们既希望利用 XML 给我们好处, 又不想在较慢的代码和较大的内存和磁盘存储方面付出代价。那么 XDS 元格式就是为解决问题而设计的。

1.13.2 简单介绍 XDS Meta 格式

XDS 最初是设计来用作一种功能全面的 3D 模型格式, 但很快就被应用于各种数据类型。XDS 是二进制数据的元格式, 大致和 XML 表达力等同。但是 XDS 虽然类似于 XML, 却不只是标记化的 XML。在这里, XML 设计主要用于可读性 (readability) 和便携性 (portability), XDS 设计主要为解析速度和最小规模服务。实际上, 它通过保留 XML 便携性和国际化能力达到两者的目标。

与 XML 相同, 在 XDS 中的所有数据都是标记的。但不同的是 XDS 标记化为两个字节的标识符, 通常是数据块惟一的花费。所以 XDS 中的数据元素是强类型的, 就如同 C/C++ 类型一样在内存中按照相同的布局存放。XDS 支持几乎 C/C++ 可能的每一种数据类型, 包括整型到枚举整型类型、字符串、固定小数点和浮点数据类型、数组、结构体, 一直到任意图

形数据类型。

XDS 文件由简短的头文件跟随许多记录组成。有记录定义数据流的名字、指定字符的编码到定义数据类型、熟悉、元素和附加的数据记录，来携带数据有效负载，支持注释和中止数据流。

XDS 中的数据记录由一个简短头文件紧接任意数目的属性、元素或者子记录。XDS 格式被优化可以使得在一个读调用（read call）中读入整条记录。

XDS 的灵活性关键在于 XDS 的数据流定义（Data Stream Definition，简称 DSD）。一个 DSD 是个仅仅包含定义一个内容流（content stream）记录的规则的 XDS 文件——也就是说它不包含任何其他数据记录。在很大程度上，一个 DSD 类似于 XML schema。首先，XML schema 定义一个 XML 文件的内容，而 DSD 则定义在一个 XDS 流中存在的类型、记录、属性和元素。其次，XML schema 是 XML 文档，而 DSD 则是一个规则的 XDS 流。与 XML 不同，保存在 XDS 文件中的数据若没有相应的数据流定义（DSD）就不能解析。

DSD 可以是 explicit，在这种情况下它能够被包括在 XDS 流中；可以是 referenced，在这种情况下它能够被从 XDS 流中单独指定但对它的一个引用被包含 XDS 在流中；可以是 implied，在这种情况下它能够被从 XDS 流中单独指定但没有任何引用在流中；或者可以是 implicit，在这种情况下它能够被编码成为读 XDS 文件的程序结构。

1.13.3 XDS 工具集



为了帮助你获得并且使用 XML 和 XDS 在你的项目 pipeline 中很快地运行，我们已经提供由两个工具和一个库组成的开发套件，外加一个 XDS 元格式的完整规格说明。这些也可以在《游戏编程精粹 4》附带的光盘或互联网上找到 [SP03]。其内容包括整个工具集的全部源代码。虽然这个工具基于 Windows 平台，但目前已开始考虑如何能更容易地将它们转换到其他平台了。

XDS 中的两个工具就是 xdsMakeSchema 和 xdsConvert。它们：

- 解析你的 C/C++ 源码收集你程序中的数据类型和变量。
- 使用收集到的数据来生成 XML schema 定义、XDS 的 DSD，以及包含 DSD 并支持

#define 的 C 语言头文件。

- 将适应 XDS DSD 的 XML 实例文档转换为 XDS 数据文件。

XDS Lite API 库也包含在 XDS 工具集中，用于：

- 读 XML 实例文档
- 读 XDS 数据文件
- 写 XDS 数据文件

图 1.13.1 描述了这两个工具、库以及它们生成的文件，这三者间的密切关系。

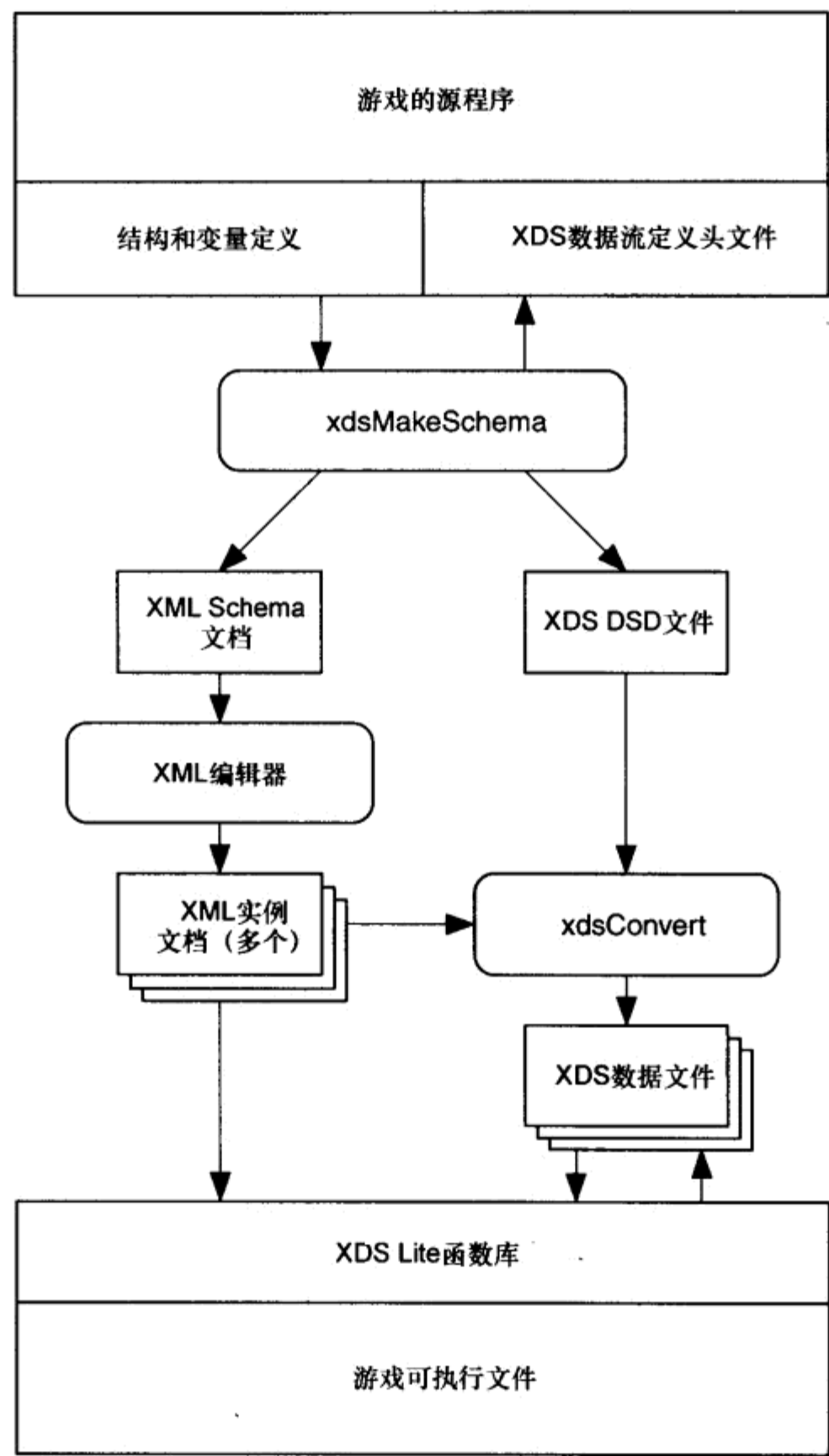


图 1.13.1 经过 XDS 工具包的数据流

1.13.4 使用 XDS 工具集

为了说明使用 XDS 实现一个系统的简单，我们将说明一步一步生成基础的 XDS 数据流的过程。例如，我们将生成并且在后面装载一个包含能力提升的定義的数据文件用于一个简单的游戏。我们已经放置了我们将要用于这个目的的数据结构，连同实例变量来包含一个叫“powerups.h”的头文件。详情如下：

```
struct PowerUp_t
{
```

```

char szName[10];      // 显示的名称
char szImage[16];     // 图像文件名

// 体力的增加/减少量 (-128~127)
signed char iHealth;

// 临时的能力增加值/惩罚减少值
// (值是以秒为单位的持续时间)
unsigned char iInvulnerability;
unsigned char iFastMove;
unsigned char iHighJump;
unsigned char iStunPlayer;

// 额外的命 (计数)
unsigned char iLifeUp;
};

// 用于定义全局 power-up 的缓冲
extern struct PowerUp_t *g_PowerUps;

```

第一步：从源代码中提取类型数据 (type data)

在我们开始编写 XML 数据文件时，我们需要一个 XML schema 定义文档。这个文档将描述你程序中的数据类型和变量以及怎样把它们组合起来。例如，这是一个 schema 中定义 power-up 结构的部分：

```

<xs:complexType name="PowerUp_t" final="#all">
  <xs:sequence>
    <xs:element name="szName">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:maxLength value="10"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
    <xs:element name="szImage">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:maxLength value="16"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
    <xs:element name="iHealth" type="xs:byte"/>
    <xs:element name="iInvulnerability"
      type="xs:unsignedByte"/>
    <xs:element name="iFastMove"
      type="xs:unsignedByte"/>
    <xs:element name="iHighJump"

```

```

        type="xs:unsignedByte"/>
    <xs:element name="iStunPlayer"
        type="xs:unsignedByte"/>
    <xs:element name="iLifeUp"
        type="xs:unsignedByte"/>
</xs:sequence>
</xs:complexType>

```

如你所见，XML schema 文档很快就会变得很大，通常它需要通过部分工作来定义一些简单的数据块。值得庆幸的是，我们无需对此担心，因为我们没有手写 (hand-writing) schema。而是我们将使用 xdsMakeSchema 工具来直接从 C/C++ 源代码中提取类型信息。

虽然 xdsMakeSchema 是有效地，但它并不聪明。它不分青红皂白地从你放进去的源代码中提取全局类型和变量定义。因为这样，你可能想在使用工具前准备好用于解析的 C/C++ 源代码。你传递到工具中的源代码应该包含你仅仅想用于作为你数据文件的主要成分的类型和变量。

xdsMakeSchema 由命令行的界面驱动。

```

xdsMakeSchema [-C] [-m macroFile]
    [-D name [= definition ]] [-U name ]
    [-s streamName] [-r recordName [: lengthSize ]]
    [-o outFile] input-files

```

假如没有提供可选参数 -s 和 -r，数据流名字将默认为 “xdsStream” 和也将有一个称为 “xdsDataRecord” 唯一的记录类型。有多个记录类型是可能的，也是常常期望得到的。未来定义附近的记录类型，仅仅增加另外的 -r 选择参数到 xdsMakeSchema 命令行。

提取类型数据和创建 XML schema 文档 (powerups.xsd)、XDS DSD (powerups.dsd) 和 XDS DSD 头文件 (powerups_dsd.h)，我们使用下面这条命令 (写成一行)：

```

xdsMakeSchema -s Powerups -r Powerup:2 -o powerups.xsd
    -o powerups.dsd -o powerups_dsd.h powerups.h

```

第二步：创建 XML 实例文档

有 XML schema 在手，现在我们来使用它。如前面提到的，XML 编辑器使用 XML schema 文档来创建一致的 XML 的实例文档。对于一个 XML 文档来说包含我们仅仅想要的数据到我们想要的格式中是随心所欲的。

假如我们正在编辑的数据过于复杂，或许可以考虑花一些时间来为数据创建一个文档模版。这是一个在很多更高级的 XML 编辑器中提供的特性允许你创建一个更加吸引人的填空表格。在一些情况下，这些表格甚至能提供一些层次的交互度。

因为提升能力 power-up 的数据是很直接的，我们将正好直接编辑它。为了做这个，我们打开 XML 编辑器并基于产生的 XML schema 来创建一个新的 XML 实例文档。当然，达到这个精确的方法将根据你使用的 XML 编辑器变化而变化；图 1.13.2 给出当我们使用 XML Spy 可能看起来的样子。

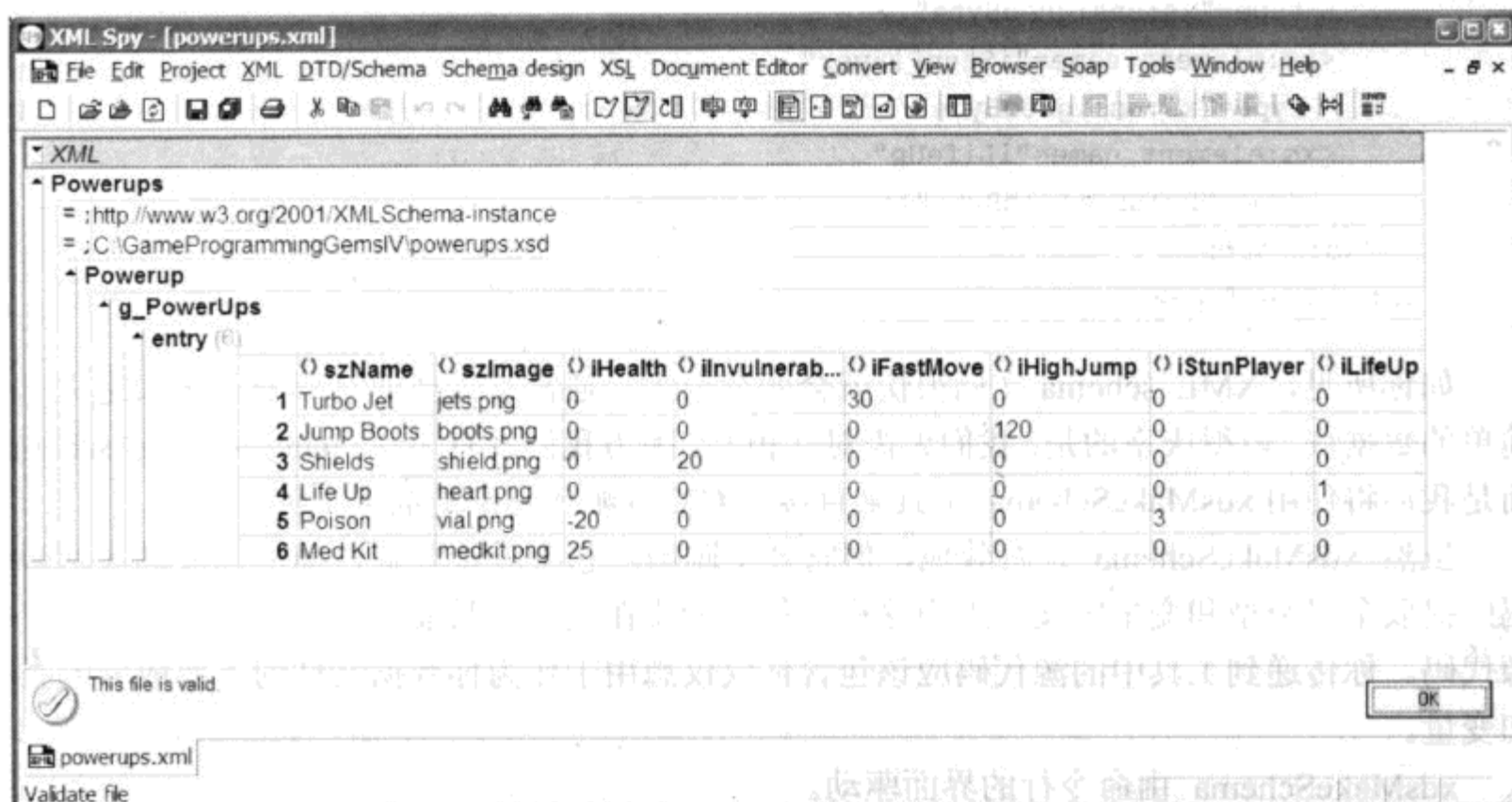


图 1.13.2 使用 XML Spy 编辑器来进行游戏数据编辑

第三步：集成 XDS Lite API 到你的游戏中

为了读取产生到游戏中的 XML 数据，就不得不编译 XDS Lite 库以及前面产生的 XDS DSD 头文件和你的游戏，然后调用到库。然而在开始动手之前，你还需要执行少量定制。

为了使库灵活，很多内容是使用“#ifdef”ed 的代码来驱动。另外，为了简化集成库到主程序的任务，库利用几个 callback 函数。因此，为了使用库，你将必须配置它和提供 callback 函数的实现。

所有库的配置选择单独在一个头文件中叫“XDSconfig.h”，但假如你在工具集中没有看到任何这样的文件，相反，却有一个样例配置头文件叫做“XDSconfig_sample.h”。那么只需要复制这个文件到“XDSconfig.h”，然后修改它来适合你的需要。下面将说明一些你需要的主要的特殊条款。

配置选项

XDS_SUPPORT_XML— 必须定义来允许读 XML 文件。

XDS_SUPPORT_WRITING— 必须定义来允许写 XDS 文件。

回调功能声明

```
int XDS_READ(void *hFile, void *buf, int iSize);
int XDS_WRITE(void *hFile, void *buf, int iSize);
void *XDS_ALLOC(void *buf, int iSize, int iMemType);
void XDS_FREE(void *buf);
void XDS_PROCESSNODE(unsigned short nodeType, void *nodeData,
    unsigned long nodeSize);
void XDS_ERROR(const char *errText);
```

在“XDSconfig_sample.h”文件中的注释探究关于这些条款的每条用于什么和期待的行

为方面的一些细节。大部分 callback 函数都相当简单且容易理解，但需要特别注意 XDS_PROCESSNODE 函数。

无论何时当一个完整数据块从一个输入流中被读入的时候，调用 XDS_PROCESSNODE 函数。传递到这个函数的数据使用 XDS_ALLOC 回调函数来分配，由主程序负责释放它。XDS_PROCESSNODE's nodeType 参数的值在 XDS DSD 头文件定义。在我们的例子中有一个我们先前生成的“powerups_dsd.h”文件。下面有这个文件中相关的几行：

```
#define XDS_Powerups_Powerup      0x0100    // 记录
#define XDS_Powerups_PowerUp_t    0x0101    // 类型
#define XDS_Powerups__xdsType1    0x0102    // 类型
#define XDS_Powerups_g_PowerUps   0x0103    // 元素
```

这些#define 名字的创建是基于以下几点：数据流的名字、记录名字和在“powerups.h”中的类型及变量。在这个例子中的“_xdsType1”是一个自动产生匿名的类型，拥有 PowerUp_t 对象的队列。这个是必须的，因为 g_PowerUps 被定义作为 PowerUp_t 的指针。

XDS_PROCESSNODE 的实现通常是一个带代码的转换声明处理每种数据类型。这是一个 power-up 样例的实现：

```
#include "powerups.h"
#include "powerups_dsd.h"

extern int g_iPowerUpCount;
void XDS_PROCESSNODE(unsigned short nodeType,
                     void *nodeData,
                     unsigned long nodeSize)
{
    switch(nodeType)
    {
        case XDS_Powerups_Powerup:
            // 记录开始——什么也不做
            break;

        case XDS_Powerups_g_PowerUps:
            // powerup 数据——保存下来
            g_PowerUps = (struct PowerUp_t *)nodeData;
            g_iPowerUpCount = nodeSize /
                sizeof(struct PowerUp_t);
            break;
    }
}
```

第四步：读数据到你的游戏中

随着这个库的配置完成，下一步是实际读数据。在所有工作做完后到这个点，装载数据就相当简单了。

```
#define DEFINE_DSD
#include "powerups_dsd.h"
```

```
FILE *fp = fopen("powerups.xml", "rb");

struct xdsHandle *h = xdsInit(fp, "Powerups",
                              &XDSDSD_Powerups[0]);

while (xdsReadRecord(h))
    ;

xdsFini(h);
```

在这里，我们使用 POSIX 的 `fopen()` 调用仅仅是为了起说明作用。你可选用不同的 I/O 系统——这正是 XDS_READ 和 XDS_WRITE 回调函数存在的原因之一。

对于 `xdsInit()` 的参数 `XDSDSD_Powerups` 包含 XDS DSD。定义在 `xdsMakeSchema` 产生的“`powerups_dsd.h`”头文件中。这个库使用它来驱动 XML 数据转换到 C/C++ 结构。

第五步：使它更快

我们的 `power-up` 的示范数据是简短的，所以从 XML 文件中读取数据也不是太糟糕。但若是放入整个关卡的游戏数据，你将注意到速度下降。下一步就是使用 `xdsConvert` 来将 XML 数据转换到 XDS，这样就不必在读的时候进行转换。

`xdsConvert` 工具由命令行的操作界面驱动。

```
xdsConvert [ -d DSD-file ] [ -x DSD | Comments |
    Signature | Name ] [ -o output-file ] input-files
```

`-x` 选项用于拒绝各种类型的数据类型写入输出流中。能够指定多个 `-x` 选项。输出文件名不能同时也是个输入文件名。如同你可能在命令语法中注意到的那样，`xdsConvert` 能够将多个输入文件接合成为一个输出文件。

为了转换我们包含 `power-up` 信息的 XML 文件，我们使用这个命令（在一行上输入）：

```
xdsConvert -d powerups.dsd -x Name -x Comments
    -o powerups.xds powerups.xml
```

在游戏中使用经过转换的 XDS 数据文件很简单，修改传给 `fopen()` 的文件名即可。XDS Lite 库通过检查文件内容来判断数据文件的类型。

第六步：使它更小

“`XDSconfig.h`”头文件有几个配置选项，它们对库的大小和速度产生影响。在这里我们将列出三个影响最大的部分加以描述。

一旦你的游戏已经足够先进，你可能想从你游戏使用 XDS Lite 库的 `copy` 中移去 XML 的支持。这个能够通过注释行为“`#define XDS_SUPPORT_XML`”来做到。这样做将单独简化库的大小和内存的需要。

假如你没计划使用库来读和写你的游戏保存文件，你也能够减小大小，通过移去所有对于写文件的支持，通过“`#define XDS_SUPPORT_WRITING`”来注释。这个也将减小库代码

大小和内存需求，虽然没有移去 XML 支持那么大。

最后，如果你想使库尽可能的精悍和快速的话，可以试着去掉行注释“`#define XDS_SUPPORT_COMPACTDSD`”。虽然这个需要和写支持一起工作，应注意，它不和所有可能提供的配置选项协调。假如它不和你选择的选项协调，在你重新编辑库的时候就会得到一个错误。


1.13.5 整合

表 1.13.1 中的指南表示在游戏开发项目中使用 XML 和 XDS 中最好的情况。通过遵守这些指南，你的项目将通过配合它们各自擅长的领域来取得 XML 和 XDS 的精华。

表 1.13.1 在项目各个阶段从 XML 移植数据到 XDS

开发阶段	数据格式与库
项目开始/原型阶段 (project start/prototype)	对于所有的数据使用 XML。使用一个双重目的的 XDS Lite 库版本来装载。最初使用标准的 XML 编辑器，直到准备特殊目的的关卡编辑器 (special-purpose level editor)
关卡设计/平衡性调整/ tweaking (level design/tuning/tweaking)	对于正在常常修改的属于使用 XML，XDS 用于稳定的资源。使用一个双重目的的 XDS Lite 库版本来装载
综合测试阶段 (integration testing)	对于所有的数据都采用 XDS。只采用仅仅 XDS 版本的 XDS Lite 库
验收测试/beta 测试/发布 (acceptance testing/beta/release)	对于所有的数据都采用 XDS。使用或者仅 XDS 版本的 XDS Lite 库或者优化的 hard-coded XDS 阅读器

1.13.6 总结



ON THE CD

XDS 元格式既快速又灵活。利用包含在附带光盘中的工具和库可让使用 XDS 变得非常容易。成组使用时，因为 XML 数据灵活且可读性强，这些工具将使你的项目有一个快速的开始，数据可以用市面上现有的编辑器编写，在发行版本中使用能够快速载入到游戏中数据结构中的 XDS 数据。

然而，提供的工具和库仅仅使用了 XDS 元格式全部功能中很小的一个子集。一个完全通用的 XDS 阅读器 (reader) 对于那些内容在首次发布后仍会继续更新的网络游戏将是特别吸引人的。这样的一个 reader，当用作精心设计的交流层的一部分时，将允许改变发送到网络的数据的类型，同时维持与早期版本的 (向上、向下) 兼容性。

1.13.7 参考文献

[Boer01] Boer, James, "A Flexible Text Parsing System," *Game Programming Gems 2*, Charles River Media, 2001.

[Olsen00] Olsen, John, "Fast Data Load Trick," *Game Programming Gems*, Charles River Media, 2000.

[SP03] "XDS Resources" Web page, available online at www.suddenpresence.com/xds.

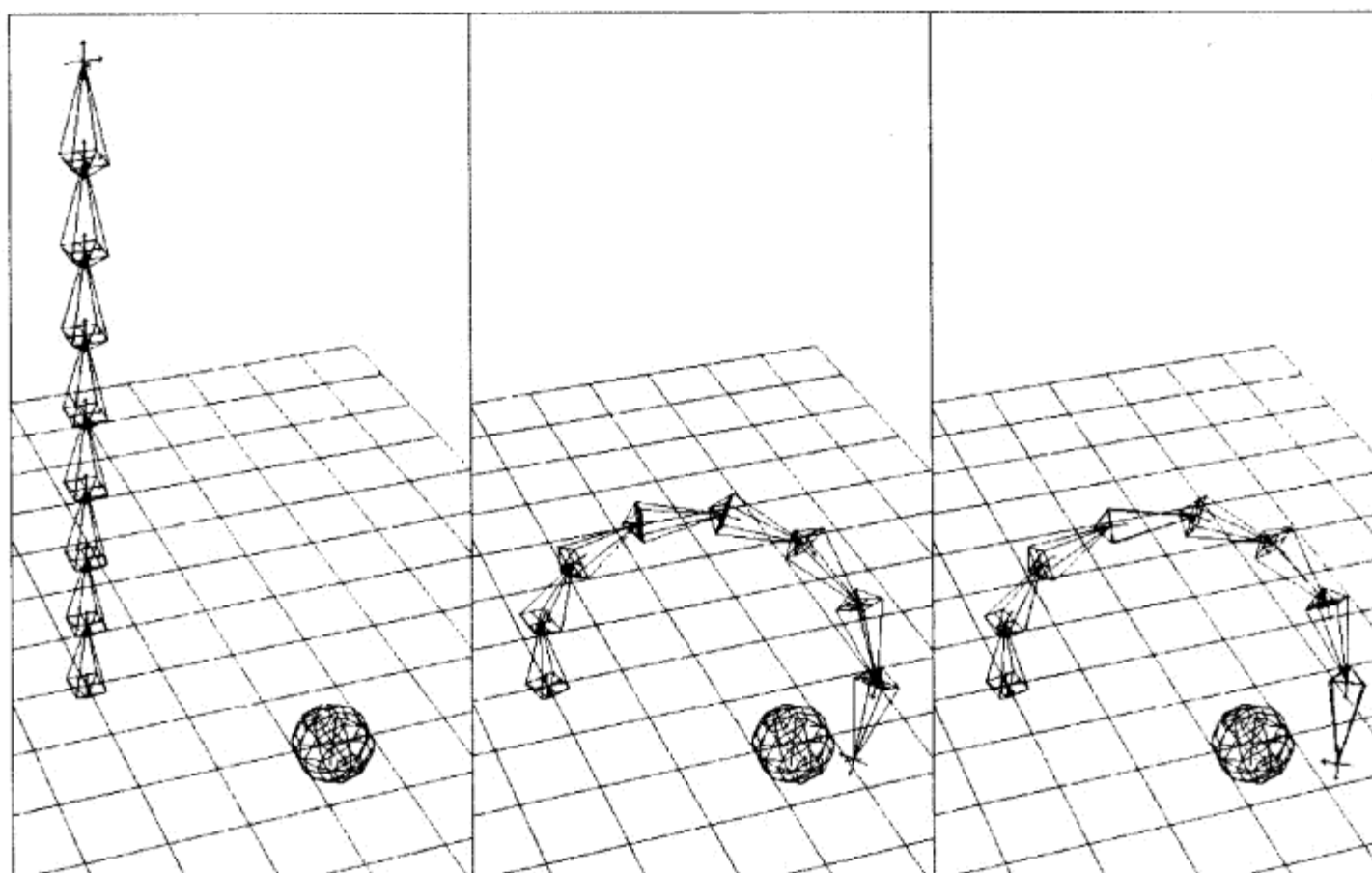
[W3C100] W3C, "Extensible Markup Language (XML) 1.0 (Second Edition)," available online at www.w3.org/TR/REC-xml, October 6, 2000.

[W3C299] W3C, "XSL Transformations (XSLT) 1.0," available online at www.w3.org/TR/xslt, November 16, 1999.

[W3C301] W3C, "XML Schema Part 0: Primer," available online at www.w3.org/TR/xmlschema-0/, May 2, 2001.



数 学



简介

作者: Jonathan Blow

E-mail: jon@number-none.com

译者: 许竹钧

审校: 沙鹰

若是在几年前,你正在读的这篇序里可能会写着“游戏程序员要出色,需要掌握不少数学知识。”这样的话,并继而举例论证。例如 John Byrd 在为《游戏编程精粹 3》的数学部分作序的时候,就给出了一些中肯而明白的例子。既然“游戏程序员应掌握相当数学知识”这一点已不再被怀疑,我的这篇序似乎应该有些新意才好。事实上,现在的游戏程序员们很想学些数学。例如,球谐函数(spherical harmonics)的使用正是今年最热门的邮件列表主题之一。

我们现在体会到要解决我们所遇到的特定的工程问题,使用数学方法是必需的。然而,我想再强调一下:由于本质和深层次的原因,数学绝对是我们工作的中心。要想术业有专攻,我们需要牢牢把握这一点,建立一种以数学为中心的方式来考虑我们的系统。

我们可以把开发游戏看成是建造一个模拟的世界。有时候这个模拟世界是抽象和离散的,比方说一个棋盘。当前(由于计算性能的提高),我们正在建造越来越多具体、连续且频繁交互的系统。我们总是在处理那些我们在真实的物理世界中熟知的概念:时间、空间和数字。

我们试图创造的是类似、甚至是模仿我们所居住的世界的小型世界。这一点让我们回想起来一篇关于物理世界的重要文章《数学之不可思议之高效》(*The Unreasonable Effectiveness of Mathematics*)。具体请参考 R.W. Hamming [Hamming80]的讨论和 Eugene Wigner [Wigner60]的论文。我不打算重复这些作者的观点,但关键在于:数学看起来是描述我们这个世界的“正确”的语言,但没人真的明白为什么会这样。如同 Hamming 所说,“总有一天,在某时某刻,我们将要解释这一现象:世界似乎由数学交织而成的逻辑模式组成,数学是科学和工程的语言。”

Hamming 和 Wigner 给出了大量的例子,比如有些数学的概念——例如“复数系统”,在没有物理用途的情况下被首次发明,并充实了某些数学领域。最后,复数的概念恰好是物理某些方面的正确描述。有些情况下,某些物理现象在被发现之前,已被数学所预知了。

由于我们出生在这科学时代,我们常常想当然地将数学和这个世界联系在一起,可当我们静下心来对我们的根基作些质疑的时候,才能看到它

究竟是多么的了不起。这会如何影响游戏创作呢？简单而大胆地利用数学之惊人的有效性，我想说有种很强的感觉让我们把这个客观世界看作是由数学构成的（可能还有其他一些东西）。

既然我们试图创造的世界是模仿由数学组成的现实世界，我们又怎可能脱离数学本身来创造这虚拟世界呢？从某种程度上说，现实世界具体表现了某些数学概念（例如导数（*derivative*）和超复数（*hypercomplex number*）），我们迟早需要掌握这些概念，才能很好地模仿这个世界。

这个观点似乎说明，较之花费我们大量时间脑力的算法和数据结构，数学要来得更加基础。“数学”有时就是我们要创造的那件东西，而算法和数据结构仅是实现的细节而已。

想象一下许多年之后的情景，游戏开发的技术水平非常先进，物理系统也大有进步。我们终于可以创造出在小尺度上几乎与客观世界一样的游戏来。这种情况下，或许可以将高级物理学家称作是“经验本体论者（*empirical ontologist*）”，而将游戏引擎的高级开发者看作是“构造本体论者（*constructive ontologist*）”。在目前而言，将游戏程序员与物理学家相提并论似乎有点不可思议。但我想，只要努力工作，假以时日，我们会到那种境界的。

游戏编程还在起步阶段，但如果将来有一天，我们所发明的高级模拟技术能帮助我们理解客观世界的一些低层行为，我也不会对此奇怪。本书的这个章节里的文章，就好比是孩童的步伐，积跬步以至千里，我们终将长大。

或者，你更愿意从短期角度看：这些文章会帮你解决一些工程问题。

参考文献

[Hamming80] Hamming, R.W., “The Unreasonable Effectiveness of Mathematics,” *American Mathematical Monthly*, Vol.87, No.2(February 1980), available online at www.lecb.ncifcrf.gov/~toms/Hamming.unreasonable.html.

[Wigner60] Wigner, Eugene, “The Unreasonable Effectiveness of Mathematics in the Natural Sciences,” *Communications in Pure and Applied Mathematics*, Vol.13, No.1(February 1960). New York: John Wiley & Sons, Inc, available online at www.dartmouth.edu/~matc/MathDrama/reading/Wigner.html.



2.1 使用马其赛特旋转的 Zobrist 散列法

作者: Toby Jones, Human Head Studios, Inc.

E-mail: tjones@humanhead.com

译者: 许竹钧

审校: 沙鹰

在很多游戏中, 同时使用深度优先和广度优先的游戏状态搜索来做决策。一旦计算出游戏的状态, 评价函数就会决定状态的优劣。这已经是一个流行的技术, 在国际象棋 (chess) [Moreland01]、围棋 (go) [Huima00] 和黑白棋 (reversi) 等游戏中都有使用, 还有很多主流的国际象棋游戏则使用了它的某个变种。这种技术也非常适合战略游戏, 特别是回合制的战略游戏。如果高层角色 AI 的离散状态集可以被限定的话, 它们在某种程度上也能使用这些技术。

通常一个好的搜索技术会产生大量需要被评估的游戏状态。由于战略游戏的特性, 游戏状态可以重复, 而且从各种不同的移动集合可以产生相同的状态。由于运行一个游戏状态的评价函数通常是一个关键的性能要因素, 所以我们要尽可能少地使用该函数。

就拿国际象棋来说吧, 我们可以将之前被评价过的每个棋盘状态的内容放入缓存。在评价任意新的棋盘状态前, 把它和已被缓存的棋盘状态列表进行比较, 如果这个棋盘在列表中已经出现过, 那么就无需再次评价。在列表中进行搜索是个线性的运算, 但使用散列可以将性能提高到具有接近常数的时间复杂度。

2.1.1 Zobrist 散列

Zobrist 散列可以从之前游戏状态的散列键值, 迅速生成 (当前) 游戏状态的散列键值 [Zobrist70]。Zobrist 散列不是散列函数的一种特定类型, 而是一种方法, 用来生成基于游戏状态并且散列效果非常好的键值。Zobrist 散列的一些关键点包括:

- 它是用简单快速的运算实现的;
- 不用重新计算整个散列就能取消运算 (undo);
- 因为类似的游戏状态产生各不相同的键值, 可以减少碰撞的发生。

实现 Zobrist 散列的确切细节随着应用程序的不同而不同, 但理念不变。它将游戏状态和代表该游戏状态参数的随机数结合在一起。结合要用

快速的数学运算来完成。运算应该是关联并可交换的，这样就可以随意地取消运算。使用随机数的话，类似的状态通常在散列键值中有几个位是不同的。

例如，要为棋盘上的小兵们生成 Zobrist 散列键值，那么给棋盘上的每个方格赋一个随机数，然后将所有的兵所在位置的值累加起来。

现在考虑移动其中的一个兵。Zobrist 散列的妙处在于，它不需要很多工作就能创建新的键值。只是从当前的键值减去当前兵所在位置的值，然后加上它在新位置的值。由于加法的特性，这就好比已经计算出完全的散列键值一样，生成了相同的键值。所以在建立搜索树的时候，这就节省了大量的时间。

2.1.2 实现 Zobrist 散列

在国际象棋中，通常使用 64 位的键值。虽然棋盘上有超过 2^{64} 种有效的位置，但可搜索的位置的数字要远远小得多，所以 64 位的键值已经足够用了。

构造一个 n 维的 Zobrist 表并用 64 位的随机数来填补。维数取决于考虑中的游戏状态参数的个数。国际象棋通常要考虑的参数为棋子的位置，类型和颜色。下一个要移动的颜色也可能是表的一个维，但由于它是游戏棋盘的一个状态，并非每个游戏棋子的状态，并不保证是一个增加的维。用其他方法来处理可能会更好些。

```
uint64_t m_aZobristTable[BOARD_SIZE]
                        [NUM_PIECES]
                        [NUM_COLORS];
```

异或函数是用来结合棋子参数的。异或的好处在于，使用相同输入，它是可逆的。遍历棋盘上所有的棋子，将把位置、棋子类型和颜色作为索引的 Zobrist 表中得到的值异或在一起。

```
uZobristKey ^= m_aZobristTable[pos][piece][color];
```

由于棋子的值是异或在一起的，移动游戏棋子的快捷方式是将以前位置的值异或到键值中去，然后将新位置的值再异或到键值中。

一旦创建了键值，如果当前玩家执黑的话，那么一个 64 位的常数就会异或到键值中。或者，改变当前玩家的时候，一直异或该常数到键值中。当要移动的一方可以用 Zobrist 表中的一个维来编码，就如棋子颜色一样，使用常数就使得要移动的一方不需要计算完整的散列键值就能被迅速定位。

一旦计算出 Zobrist 的键值，将它作为散列键值使用的最简单的方式就是：作为一个地址对散列表的大小取模。由于碰撞的几率足够小，所以在实际使用中有时会干脆忽略碰撞，如此虽然牺牲了一定健壮性，但却获得了速度的优势。一般说来，当使用 Zobrist 散列时，引起碰撞的主要原因是散列表太小了。在物理内存许可的情况下，散列表应该尽可能的大，但不要超出限度，要避免使用虚拟内存。

让散列表碰撞最少的原因是和在 Zobrist 表中的数字选择有关。由于使用了随机数，移动一个游戏棋子会产生一个完全不同的 Zobrist 键值。虽然可以仔细的挑选这些数字，但使用高度随机的数字一样有效。C 语言中的 rand() 函数经常被用来做生成 Zobrist 表中的值。然而，

rand()函数实际上不是那么随机,而且它仅生成 15 位的数字。因此,我们倾向于使用快速、64 位的“更随机的”函数。

2.1.3 马其赛特旋转 (Mersenne Twister)

马其赛特旋转是个周期超长而且分布高度均匀的快速伪随机数生成器。尽管没有伪随机的数学定义,马其赛特旋转还是能满足大量的测试,包括 k 分布测试,这是一个被公认为很强的伪随机测试。

马其赛特旋转有一些很好的特性。

- 它的实现很容易适应于 64 位数字的生成。
- 623 维均匀分布的属性保证了生成数字的所有的位都是相当随机的。
- 它的周期超长,为 $2^{19937} - 1$,也就是在出现重复之前的长度。
- 它的实现是用简单、快速的运算来完成的,同时考虑了缓存的使用,并且许多实现比 rand()要快得多。

对于任何常规的随机数生成器而言,这些都是很好的属性,但作为和 Zobrist 散列一起使用的话,前两个属性则是关键。

以前的生成器在很多地方有缺陷,马其赛特旋转就是为了解决其中的一些缺陷而特别设计的。它基于线性回归 (linear recurrence) 的原理。

$$x_{k+n} = x_{k+m} \oplus (x_k^u | x_{k+1}^l) A \quad (2.1.1)$$

x 是一个 w 位的字。 x_k^u 代表了 x_k 的前面 $w-r$ 的位, x_{k+1}^l 代表了 x_{k+1} 的后面 r 位。异或和连接分别对应于 \oplus 和 $|$ 。 w 、 r 、 m 、 n 和 A 都是常量参数。

A 是一个 w 乘 w 的矩阵表:

$$\begin{pmatrix} 0 & I_{w-1} \\ a_{w-1} & a \end{pmatrix} \quad (2.1.2)$$

I_{w-1} 是个单位矩阵, a 是代表 a_{w-2}, \dots, a_0 的向量。尽管马其赛特旋转的所有参数都会对所生成数字的属性产生影响,之所以选择该矩阵,是因为可以用快速的位运算来完成该实现。

对每个生成的字都进行下列转换,来改善 k 分布。

$$\begin{aligned} y &= x \oplus (x \gg u) \\ y &= y \oplus ((y \ll s) \cap b) \\ y &= y \oplus ((y \ll t) \cap c) \\ y &= y \oplus (y \gg l) \end{aligned} \quad (2.1.3)$$

u , s , t , l , b 和 c 都是马其赛特旋转实现的常量参数。 \ll 和 \gg 分别是左移和右移符, \cap 是代表 AND 的位运算符。

马其赛特旋转是个参数化的算法,参数如何设定则决定随机数的特性。推荐的一个实现是 MT19937,它对参数作如下定义:

$$\begin{aligned}
 w &= 32, n = 624, m = 397, r = 31, \\
 u &= 11, s = 7, t = 15, l = 18, \\
 A &= 0x9908B0DF, \\
 b &= 0x9D2C5680, \\
 c &= 0xEFC60000
 \end{aligned}
 \tag{2.1.4}$$

2.1.4 马其赛特旋转的实现

开始时, 用 32 位的数字作为长度为 624 的数组 `s_aMT` 的种子。除零之外的任何数字都是有效的。在 [Matsumoto98] 中的参考实现使用了一个有种子的随机数生成器, 但作者喜欢用质数来对数组进行静态初始化。由于所有的种子有相同周期, 所以不存在问题。

用 `m_ix` 作为 `s_aMT` 的索引, 一个 32 位的随机数是这样来计算的:

```

static const int MT_W = 32;
static const int MT_N = 624;
static const int MT_M = 397;
static const int MT_R = 31;
static const uint32_t MT_A = 0x9908b0df;
static const int MT_U = 11;
static const int MT_S = 7;
static const uint32_t MT_B = 0x9d2c5680;
static const int MT_T = 15;
static const uint32_t MT_C = 0xefc60000;
static const int MT_L = 18;
static const uint32_t MT_LLMASK = 0x7fffffff;
static const uint32_t MT_UMASK = 0x80000000;

uint32_t y;
y = s_aMT[m_ix++];
y ^= y >> MT_U;
y ^= y << MT_S & MT_B;
y ^= y << MT_T & MT_C;
y ^= y >> MT_L;

```

当耗尽数组中的所有 624 个数字后, 数组需要被重新载入。这可能是马其赛特旋转最复杂的部分。

```

for(int kk = 0; kk < MT_N; kk++)
{
    uint32_t ui = (s_aMT[kk] & MT_UMASK) |
        (s_aMT[(kk + 1) % MT_N] & MT_LLMASK);
    s_aMT[kk] = s_aMT[(kk + MT_M) % MT_N] ^
        (ui >> 1) ^ ((ui & 0x00000001) ? MT_A : 0);
}

```

在实际实现中, 为了减少取模操作符, 循环通常比较松散。

默认情况下, 马其赛特旋转生成了 32 位的数字, 但依据 [Matsumoto98] 建议, 连接两

个连续的 32 位数字来创建一个 64 位数字是有效的。

虽然马其赛特旋转有很多不错的使用,包括噪音生成或者作为 `rand()` 的快速替换,但它不是非常强的。由于它的高度均匀分布,对生成需要高度随机数字的 Zobrist 表而言,它是个不错的选择。

2.1.5 结论

Zobrist 散列的使用很好地减少了对游戏状态重复评价的次数。这为更加复杂的评价函数和更巧妙的 AI 开启了大门。因为马其赛特旋转既快又可靠,所以它是 `rand()` 的一个很好的替换。算法和 Zobrist 散列一起使用时效果很好,但它也可以在别的应用中单独使用。

2.1.6 参考文献

[Huima00] Huima, Antti, "A Group-Theoretic Zobrist Hash Function," available online at <http://persoweb.francenet.fr/~fgrieu/zobrist.pdf>, December 31, 2000.

[Matsumoto98] Matsumoto, Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator," available online at www.math.keio.ac.jp/~matumoto/emt.html, January 1998.

[Moreland01] Moreland, Bruce, "Computer Chess," available online at www.seanet.com/~brucemo/chess.htm, 2001.

[Zobrist70] Zobrist, A. L., "A New Hashing Method with Application for Game Playing," Technical Report 88, University of Wisconsin, April 1970.



2.2 抽取截锥体和 camera 信息

作者: Waldemar Celes, Computer Science Department, PUC-Rio

E-mail: celes@inf.puc-rio.br

译者: 许竹钧

审校: 沙鹰

为了实现一些图形算法, 我们需要关于物体空间中的视图截锥 (view frustum) 和 camera 参数方面的信息。例如, 一个挑选 (culling) 算法要计算在包围体 (bounding volume) 和锥平面的交集。一个多分辨率算法需要访问 camera 参数来计算相应的细节层次。就算一个简单的定位广告牌的算法都取决于视图参数 (可能需要视图及向上的方向)。如果我们想要设计独立于任何图形引擎的算法, 那么我们会面临无法追踪 camera 位置和方向的问题, 这样就会使抽取视图信息的工作非常困难。幸好, 还有一种简单而又直接的方式来抽取截锥体和 camera 信息, 它基本上是基于变换 (建模、视图和投影) 矩阵来实现的。

在本文中, 我们给出了如何简单抽取视图信息的细节。该讨论会因为潜在的图形应用程序接口 (API) 而有些改变。这里, 我们首先采用 OpenGL (开放性图形语言) 的规定; 然后将做些改动, 将其用于其他的 API, 并将它扩展到随意的投影变换。

2.2.1 平面变换 (Plane Transformation)

在经过渲染管线 (rendering pipeline) 的几何阶段, 一个定点要变换到不同的空间或坐标系统, 从物体空间到裁剪空间 [Akenine-Möller02]。图 2.2.1 用相应的变换矩阵, 说明了一个典型的渲染管线的空间序列。该视图截锥变换到了一个在裁剪空间代表规范视图盒 (canonical view volume) 的边界对齐盒 (axis-aligned box)。

乘以相应的矩阵 M , 顶点 V 就变换到每个不同的空间, 所以变换后的顶点 $V' = MV$ 。用相同的矩阵可以变换多边形, 实际上, 变换多边形的所有顶点就可得到变换后的多边形。

然而, 这个矩阵 M 并不能一直用来变换平面 (还有法矢量)。让我们看一个简单例子, 如图 2.2.2 所示, 一个单元立方体要做一个切变 (shear transformation)。如果我们用这个切变矩阵来变换立方体的右平面的话, 会得到错误的结果 (见图 2.2.2 (b))。产生的法矢量不再和相应的表面相垂直。那应该用哪一个矩阵来变换平面 (还有法矢量) 呢?

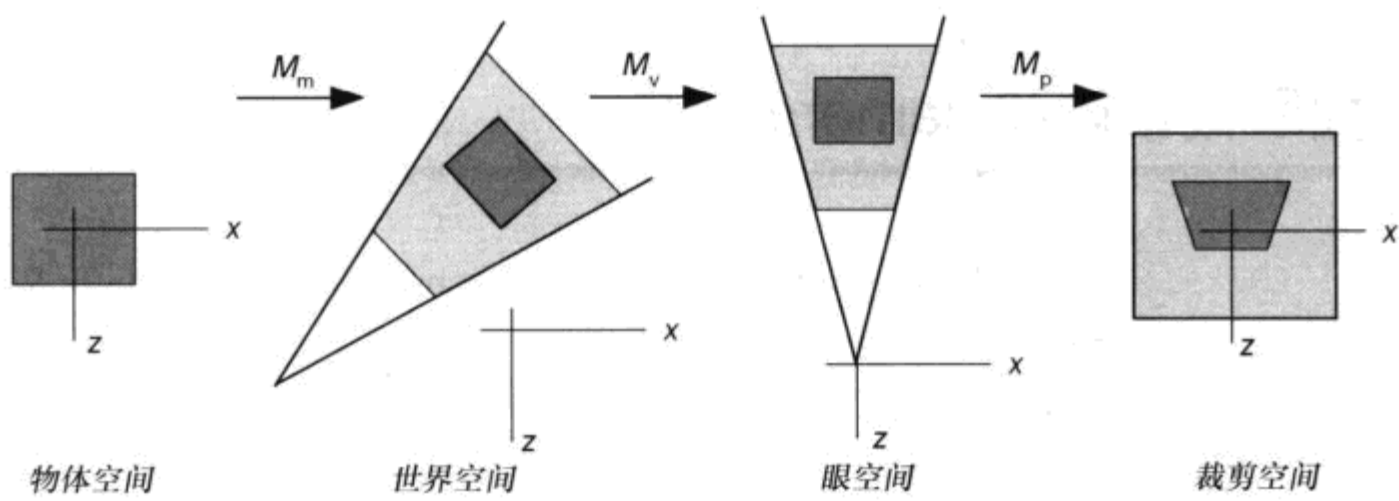


图 2.2.1 沿着渲染管线的空间和变换。在 OpenGL 中，为了性能方面的原因，模型 M_m 和视图 M_v ，及其变换，是用一个简单、已累积的模型视图矩阵（modelview matrix） M_{mv} 来表示的

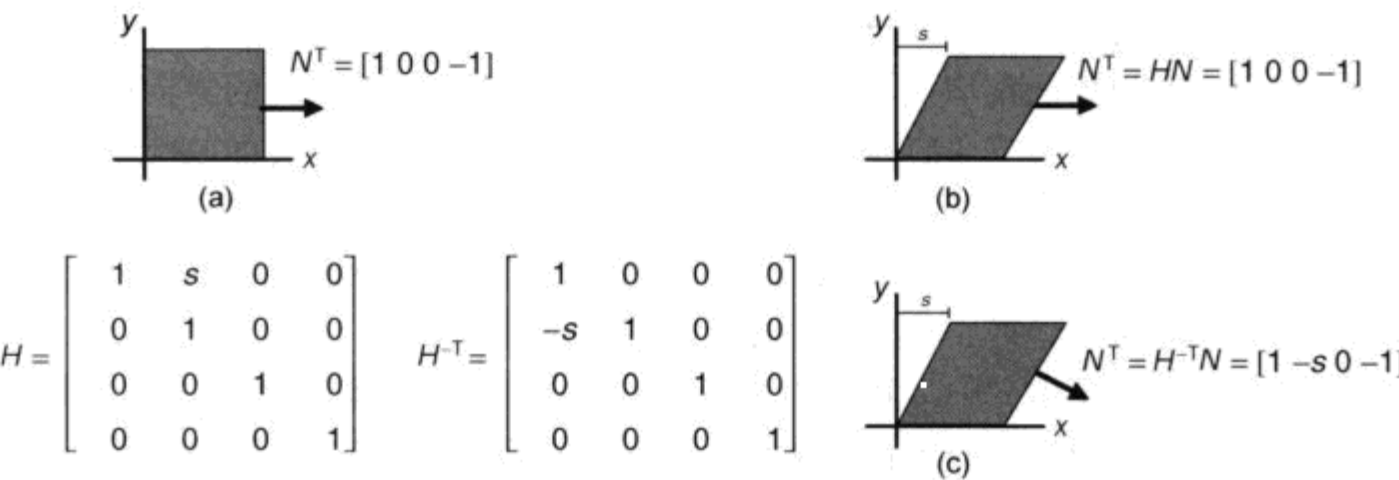


图 2.2.2 平面上的切变效果 H ：
(a) 原始模型 (b) 不正确的平面变换 (c) 正确的平面变换

可以看到用相应的逆转倒置矩阵（inversetranspose）可以变换平面。这在[Foley96]已经做了说明。平面公式如下：

$$ax + by + cz + d = 0 \tag{2.2.1}$$

如果我们用一个 4 维的行向量来表示平面公式的系数， $N^T=[a \ b \ c \ d]$ ，并用齐次坐标来表示那个平面上的点， $P^T=[x \ y \ z \ 1]$ ，那么我们可以用下面的向量形式来改写平面公式：

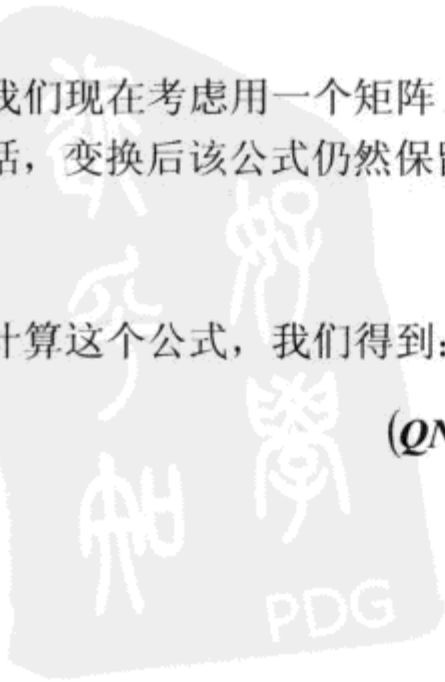
$$N^T P = 0 \tag{2.2.2}$$

我们现在考虑用一个矩阵 M 来变换点 P ，而且我们想要找到相应的矩阵 Q ，它来变换平面的话，变换后该公式仍然保留。

$$N'^T P' = (QN)^T (MP) = 0 \tag{2.2.3}$$

计算这个公式，我们得到：

$$(QN)^T (MP) = N^T Q^T M P = N^T (Q^T M) P \tag{2.2.4}$$



因此, 如果 $Q^T M = I$, 我们得到 $N^T(QTM)P = N^TIP = N^TP = 0$, 这样我们能得到 $Q = (M^{-1})^T = M^{-T}$ 。所以, 如果一个矩阵包含随意的变换, 只要它是可逆的 (就是说, 非奇异的), 用逆转倒置矩阵就可以变换该平面。图 2.2.2 (c) 给出了应用切变矩阵的逆转倒置得到的正确变换平面。

如果只考虑法矢量, 我们得到 $N^T = [a \ b \ c \ 0]$, 同时只需要考虑矩阵左上方的 3×3 部分 [Turkowski90]。在这个例子中, 矩阵可以包含不保留剪切或不规则角的变换, 但不可以包含透视变换 (perspective transformation)。此外, 如果顶点矩阵完全由如切体变换 (rigid-body transformation) 和均匀比例组成, 由于相同的顶点矩阵适用于平面, 所以不需要做特殊处理。

用逆转倒置矩阵变换平面, 实际上可以帮助我们基于变换平面来抽取原始平面。如果, 用矩阵 Q 让平面移到被变换的空间, 我们应用逆转矩阵, Q^{-1} , 把平面送回原始空间。因此, 只要得到用来变换顶点的矩阵 M , 我们就可以用它的倒置来做平面变换 ($Q^{-1} = M^T$)。

2.2.2 抽取锥体信息

视图截锥是用 6 个平面来定义的, 左、右、底、顶、近和远。这些平面的二维视图在图 2.2.3 中做了说明。如果我们在裁剪空间里有这些平面, 那么, 我们通过原来用作变换顶点的矩阵的倒置, 就能很轻松地把它变换回物体空间。

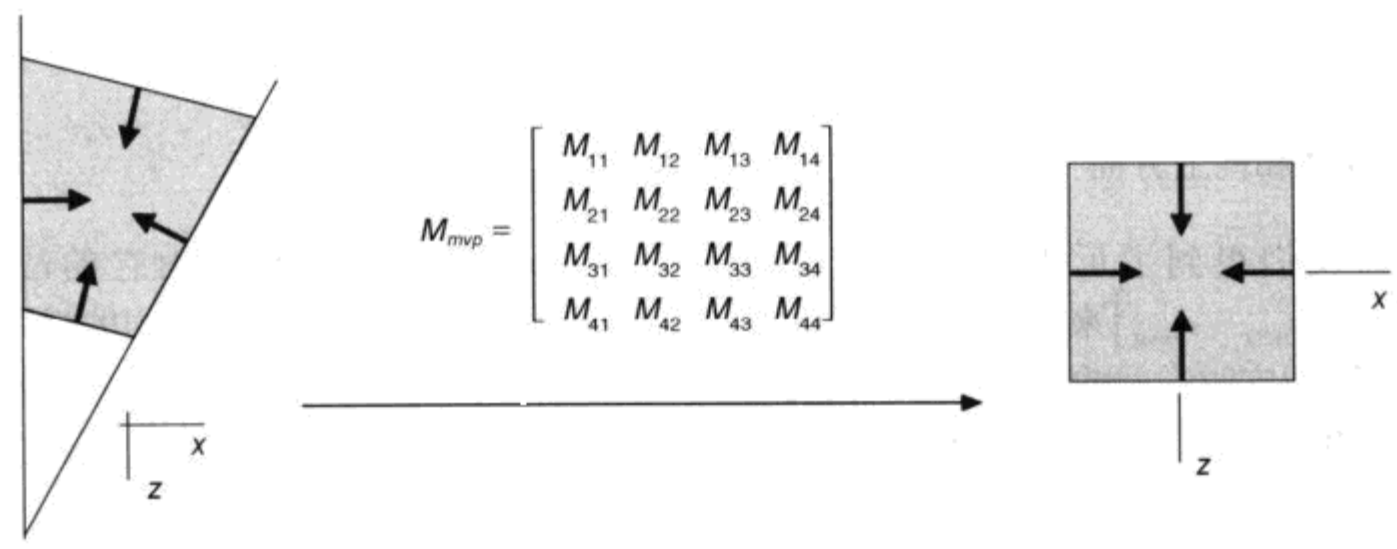


图 2.2.3 顶点变换矩阵以及在物体和裁剪空间的锥体平面

用 OpenGL 的术语, 我们研究一下模型视图矩阵 M_{mv} , 和投影矩阵 (projection matrix) M_p 。如果我们将两者合并, 那么我们最后就得到一个单独的矩阵, 模型观察投影矩阵 (modelview-projection matrix), $M_{mvp} = M_p M_{mv}$, 它把顶点从物体空间变换到裁剪空间 (见图 2.2.3)。相反地, 倒置矩阵则把平面从裁剪空间变换到物体空间。

在裁剪空间的规范视图盒代表了变换后的视图截锥。因此, 物体空间的视图截锥平面可以从相应的规范盒中得到。可以用模型观察投影矩阵乘以平面公式的系数矢量。此外, 我们更不需要计算整个的矩阵矢量乘法。由于规范视图盒是用边界对齐盒来表示的, 裁剪空间的平面公式相当简单, 我们可以直接从矩阵块来说明物体空间的平面。

我们来考虑左平面。在 OpenGL 中, 规范盒是用对角顶点在 $(-1, -1, -1)$ 和 $(1, 1, 1)$ 的立方体来表示的。因此, 在裁剪空间中, 左平面的法矢量用 $(a', b', c') = (1, 0, 0)$ 表示。知道属于该平面的点 $(-1, 0, 0)$, 就可以得到独立平面的系数。在平面公式作个替换, 我们

得到: $1(-1) + d' = 0$, 也就是 $d' = 1$ 。在裁剪空间的左平面就是 $N'^T = [1 \ 0 \ 0 \ 1]$, 它由倒置矩阵变换而来, 并得到:

$$N = M_{mvp}^T N' = [M_{11} + M_{41} \quad M_{12} + M_{42} \quad M_{13} + M_{43} \quad M_{14} + M_{44}]^T \quad (2.2.5)$$

类似地可以得到其他平面。物体空间的 6 个锥体平面是用下面的系数来表示。在[Gribb01, Akenine-Möller02], 用一种不同的方式可以得到相同的结果。

$$\begin{aligned} N_{\text{left}} &= [M_{41} + M_{11} \quad M_{42} + M_{12} \quad M_{43} + M_{13} \quad M_{44} + M_{14}]^T \\ N_{\text{right}} &= [M_{41} - M_{11} \quad M_{42} - M_{12} \quad M_{43} - M_{13} \quad M_{44} - M_{14}]^T \\ N_{\text{bottom}} &= [M_{41} + M_{21} \quad M_{42} + M_{22} \quad M_{43} + M_{23} \quad M_{44} + M_{24}]^T \\ N_{\text{top}} &= [M_{41} - M_{21} \quad M_{42} - M_{22} \quad M_{43} - M_{23} \quad M_{44} - M_{24}]^T \\ N_{\text{near}} &= [M_{41} + M_{31} \quad M_{42} + M_{32} \quad M_{43} + M_{33} \quad M_{44} + M_{34}]^T \\ N_{\text{far}} &= [M_{41} - M_{31} \quad M_{42} - M_{32} \quad M_{43} - M_{33} \quad M_{44} - M_{34}]^T \end{aligned} \quad (2.2.6)$$

2.2.3 抽取 camera 信息

一旦我们得到锥体平面, 所有基本的 camera 参数都可以马上用计算交叉乘积和平面交集来得到。我们也能用模型视图矩阵从眼空间到物体空间来抽取信息。

1. 视图和向上方向

一旦我们得到了近平面, 那么视图方向 (view direction) 直接由它的法矢量: $V_{\text{dir}}^T = [a_{\text{near}} \ b_{\text{near}} \ c_{\text{near}}]$ 来给定。由于模型视图矩阵不包括透视变换, 所以相同的视图方向也可以用先取得它眼空间的值, 再带回到物体空间的办法来获取。在眼空间中, 视图方向是以 $V_{\text{dir}}'^T = [0 \ 0 \ -1]$ 来给定的, 结果用 $V_{\text{dir}}^T = -[M_{31} \ M_{32} \ M_{33}]$ 来表示, 其中 M 代表模型视图矩阵。

向上方向也可以用两种方式抽取。我们可以用评估左右锥体平面法矢量的交叉乘积来计算, $V_{\text{up}}^T = [a_{\text{left}} \ b_{\text{left}} \ c_{\text{left}}] \times [a_{\text{right}} \ b_{\text{right}} \ c_{\text{right}}]$, 或是类似于视图方向, 由于向上方向在眼空间中以 $V_{\text{up}}'^T = [0 \ 1 \ 0]$ 来表示, 我们可以用模型观察来计算在物体空间中的向上方向。因此, 它在物体空间中可以用 $V_{\text{up}}^T = [M_{31} \ M_{32} \ M_{33}]$ 来表示, 同样地, M 代表模型视图矩阵。

如果我们需要视图和向上方向矢量都被规一化的话, 那么就应该谨慎。如果视图方向是从规一化后的锥体平面中抽取的, 那么所得到的矢量已经是规一化的, 否则, 就需要做显示的规一化。对于向上方向而言, 则从头到尾需要规一化。

2. Camera 位置

一些算法需要知道观察者的位置来执行运算。可惜的是, 它是从变换矩阵中所要抽取的最昂贵的信息。再次说明, 我们可以用两种方式来抽取: 通过锥体平面或是通过模型视图矩阵。要注意, 这信息只对透视投影有意义, 因为对于正投影 (Orthographic Projection) 而言, 观察者是无限放置的, 并且信息只和视图方向相关。

基于锥体平面, 我们可以计算 3 个平面的交集来抽取 camera 信息, 例如左、右和顶平面。我们就得到一个要解答的线性公式系统 $AP=B$, 其中

$$A = \begin{bmatrix} a_{\text{left}} & b_{\text{left}} & c_{\text{left}} \\ a_{\text{right}} & b_{\text{right}} & c_{\text{right}} \\ a_{\text{top}} & b_{\text{top}} & c_{\text{top}} \end{bmatrix}, \quad P = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \quad B = \begin{bmatrix} -d_{\text{left}} \\ -d_{\text{right}} \\ -d_{\text{top}} \end{bmatrix} \quad (2.2.7)$$

使用 Cramer 规则, 我们最后要计算一些矩阵行列式。

$$P = \begin{bmatrix} \det A_a / \det A \\ \det A_b / \det A \\ \det A_c / \det A \end{bmatrix} \quad (2.2.8)$$

这里, A_i 是第 i -列被 B 替换后的矩阵 A 。如果 $\det A$ 等于 0, 那么它意味着观察者正处于无穷中 (正投影)。

当然, 也可以从模型视图矩阵得到 camera 位置。由于我们知道在眼空间中的 camera 位置, 可以应用模型视图矩阵的逆转来得到在物体空间的位置 (这里, 我们需要逆转矩阵是因为我们正在变换一个点)。然而, 我们不需要计算整个逆转矩阵。在眼空间中, camera 在初始位置, 用 $P^T = [0 \ 0 \ 0 \ 1]$ 的齐次坐标来表示。因此, 对于 $P^T = [M_{31}^{-1} \ M_{24}^{-1} \ M_{34}^{-1}]$, 它完全能找到逆转矩阵的第 4 列, 其中, M^{-1} 表示模型视图矩阵的逆转。

3. 近和远距离

知道物体空间中的近、远平面距离也许是必要的。在物体空间中, 一旦我们已经规一化 camera 位置和近、远平面, 那么用评价平面的公式就可计算出之间的距离。

$$\begin{aligned} D_{\text{near}} &= -a_{\text{near}}P_x - b_{\text{near}}P_y - c_{\text{near}}P_z - d_{\text{near}} \\ D_{\text{far}} &= a_{\text{far}}P_x + b_{\text{far}}P_y + c_{\text{far}}P_z + d_{\text{far}} \end{aligned} \quad (2.2.9)$$

如果模型视图矩阵不包括非刚体变换 (non-rigid body transformation), 那么我们可以在眼空间中计算这些距离。在那种情况下, 使用投影矩阵, 我们可以将近远平面从裁剪空间带到眼空间。距离可以用 $D_{\text{near}} = -d_{\text{near}}$ 和 $D_{\text{far}} = d_{\text{far}}$ 来给定, 其中 d 代表眼空间中规一化平面公式的独立组件。

4. 视角

视野的垂直和水平角度, θ_v 和 θ_h , 也可以从锥体平面计算得到。如果我们在物体空间对平面已经作了规一化, 只要简单地评价点乘积就可抽取到角度。

$$\begin{aligned} \cos(\pi - \theta_v) &= [a_{\text{bottom}} \ b_{\text{bottom}} \ c_{\text{bottom}}] \cdot [a_{\text{top}} \ b_{\text{top}} \ c_{\text{top}}] \\ \cos(\pi - \theta_h) &= [a_{\text{right}} \ b_{\text{right}} \ c_{\text{right}}] \cdot [a_{\text{left}} \ b_{\text{left}} \ c_{\text{left}}] \end{aligned} \quad (2.2.10)$$

如果锥体是非对称的 (在虚拟现实应用中很常见), 那我们不得不在每个方向中计算两个角度。在这个例子中, 我们可以在计算中使用近平面的法矢量。比如说, 视图水平向左的角度可以用在近和左锥体平面之间的点乘积计算得到。

2.2.4 任意投影变换

用其他的图形 API 也能得到相同的结果。也可以在裁剪空间考虑合适的平面公式。在 DirectX 中，例如，近远平面用 $z = 0$ 和 $z = 1$ 来分别给定。所以其他的导出信息也可以类似地抽取到。

我们可以将这个讨论进一步深化，归纳到任意的投影变换。已知所有的三维投影变换是用 4×4 的非奇异矩阵来表示的（并且每个 4×4 的非奇异矩阵是投影变换）[Penna86], [Davis01]。此外，如果我们可以来来回回地将一个点变换到不同空间，那么我们也能变换平面，这是因为点和平面是三维投影空间的双重物体。

因此，只要得到相应的顶点变换矩阵，我们就可以把任何平面带回它的原始空间。例如，如果已经累积了映射到变换矩阵的屏幕，那么我们就能从屏幕空间抽取视图截锥的信息。

当然，我们不得不根据相应变换的意义来适应所抽取信息的语义。“camera 位置”实际上是透视锥体的顶点。为了说明其他的用法，我们来研究用来计算阴影的透视变换，这是由 Heckbert 和 Herf [Heckbert97] 提出的，在[Akenine-Möller02]中有介绍。这个透视变换将一个以平行四边形为底的锥体映射到一个边界对齐盒。锥体的顶点对应于光的位置，平行四边形底对应阴影接收部分。锥体的左右两个面变换到平面 $x = 0$ 和 $x = 1$ ，同时顶和底分别变换到 $y = 0$ 和 $y = 1$ 。锥体底平面变换到 $z = 1$ ，经过顶点并平行于它的一个平面则变换到 $z = \infty$ （图 2.2.4）。

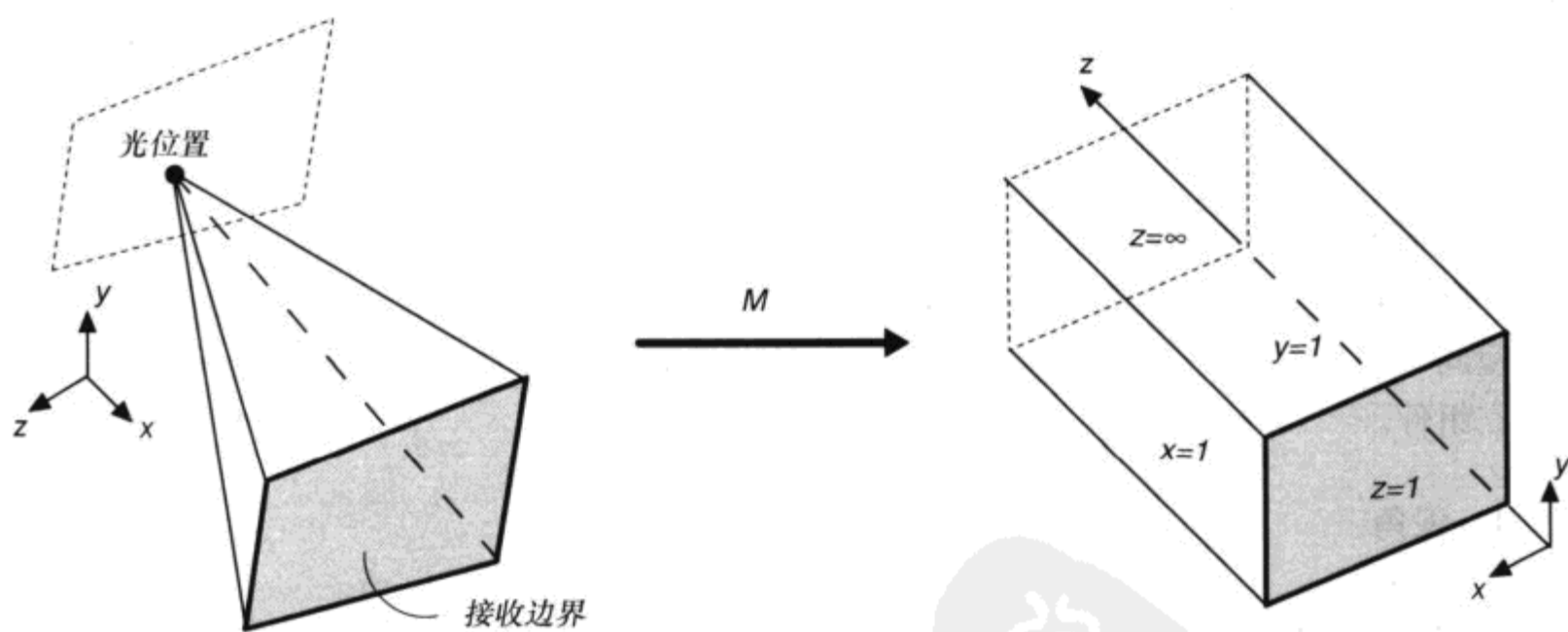


图 2.2.4 平行四边形为底的锥体的透视变换

基于该类型的一个透视矩阵，我们可以恢复诸如光位置和接收边界的有用信息。为了抽取该类信息，我们首先将平面带回物体空间，要记住的是处于无限位置的平面在透视几何中是用 $[0 \ 0 \ 0 \ 1]^T$ 来表示的。然后，计算左、右还有顶平面的交集可以取得光位置。由于接收边界的顶点代表了近（底）平面和锥体侧面的交集，用类似的方法可以取得。

2.2.5 实现



配套光盘包含了基于 OpenGL 变换矩阵来抽取锥体和 camera 信息的 C++ 代码。对于该实现，为了保持代码的一般性，我们选用基于锥体平面来抽取 camera 信息。对于特定的应用，可能用别的（如之前的章节所讨论的方法）来计算，会更有效些。代码也提供了从随意投影变换抽取信息的基类。

1. 类的构造函数

代码实现了一个 VglFrustum 类，它提供了从 OpenGL 矩阵查询锥体和 camera 信息的方法。我们可以传递用来抽取视图信息的相应变换矩阵，以此来创建那个类的对象。它提供了三种不同的构造器：

```
VglFrustum (float* Mmv, float* Mp);  
VglFrustum (float* Mmvp);  
VglFrustum ();
```

和在 OpenGL 中一样，矩阵使用矢量来表示的，一列接一列。如果我们同时得到模型观察和投影矩阵的话，最好用第一个构造器。这两个矩阵然后累积成一个单独的模型观察投影矩阵，可以用它来从裁剪到物体空间中抽取信息。如果我们已经有了累积的矩阵，最好用第二个构造器。如果我们想要在眼空间中抽取信息的话，这个构造器也很有用；在那种情况下，我们只需要提供投影矩阵。最后一个构造器用查询图形 API 的当前状态来抽取矩阵。由于它使得在中央和图形处理器之间强加了同步，所以在任何时候，都要避免它在渲染管线内的使用。

2. 查询方法

一旦创建了一个对象，那我们就可以抽取锥体和 camera 信息。对于查询锥体的信息，则给出了如 GetPlane() 的方法，它返回了在物体空间的相应平面系数。返回的平面公式不是自动规一化的——只要需要，就应该显示地请求规一化。对于抽取 camera 信息，则有如 GetViewDir() 和 GetEyePos() 等方法。

3. 基类

类 VglFrustum 是在类 AlgFrustum 之上构建的，后者是用来从随意投影变换中抽取平面公式。这个类提供了一个静态方法 SetCanonicalPlane(), 用来在变换的空间中报告相应的平面公式。此外，GetPlane() 能使我们在对象空间中恢复这样的平面。一旦得到了平面公式，我们就可以用在类 AlgPlane 和 AlgVector 中，诸如 Intersect() 和 Cross() 等有效方法，来抽取位置 and 方向。派生一个适合 DirectX API 的 Frustum 类是很简单的。

2.2.6 结论

一些图形算法需要抽取锥体和 camera 信息来执行计算。本文说明了如何从有效并且实用的角度来抽取此类信息，也对如何从一般的投影变换抽取信息展开了讨论。很好地理解此类变换对于游戏程序员而言是非常有用的，因为他们可以选择最好的空间来执行特定的计算。

2.2.7 参考文献

[Akenine-Möller02] Akenine-Möller, Tomas and Eric Haines, *Real-Time Rendering, Second Edition*, A. K. Peters, 2002.

[Davis01] Davis, Tom, "Homogeneous Coordinates and Computer Graphics," *Mathematical Circles*, available online at www.geometer.org/mathcircles, November 20, 2001.

[Foley96] Foley, James, Andries van Dam, et al., *Computer Graphics, Principles and Practice, Second Edition*, Addison-Wesley, 1996.

[Gribb01] Gribb, Gil and Klaus Hartmann, "Fast Extraction of Viewing Frustum Planes from World-View-Projection Matrix," available online at www2.ravensoft.com/users/ggribb/plane%20extraction.pdf, June 15, 2001.

[Heckbert97] Heckbert, Paul S. and Michael Herf, "Simulating Soft Shadows with Graphics Hardware," Technical Report CMU-CS-97-104, Carnegie Mellon University, available online at www.cs.cmu.edu/~ph/shadow.html, January 1997.

[Penna86] Penna, Michael A. and Richard R. Patterson, *Projective Geometry and its Applications to Computer Graphics*, Prentice-Hall, 1986.

[Turkowsky90] Turkowski, Ken, "Properties of Surface Normal Transformations," in Glassner, Andrew, *Graphics Gems*, Academic Press, available online at www.worldserver.com/turk/computergraphics/index.html, 1990.



2.3 解决大型游戏世界坐标中的精度问题

作者：Peter Freese, NCsoft Core, Technology Group
E-mail: pfreese@ncaustin.com
译者：许竹钧
审校：沙鹰

由于浮点数精度方面的问题，创建超大型连续游戏世界的开发者总是不可避免地陷入到麻烦中。几乎每个游戏平台都使用 32 位浮点数来表示坐标，然而当坐标值较大而同时要求没有明显的精度错误时，浮点数就没法表示比第一人称视角射击游戏的场景更大的区域。本文介绍了一种能处理大型坐标的技术，它快速高效，同时也不需要切换到双精度数。

在这种技术中，通常由三个浮点数部分组成的全局空间位置，被增多了个整型数 segment（片），它有 offset（偏移）；或者说传统位置，能被规一化到一个期望的精确范围。“segment”和“offset”是从分段处理器地址空间符借鉴而来，而诸如重新规一化等许多概念，出处类似。

2.3.1 问题描述

对于二进制浮点算术，IEEE 754 定义了什么是通常所说的 IEEE 浮点数 [Goldberg91]。在单精度 32 位 IEEE 格式中，1 位是符号位，接下来的 8 位是指数部分，剩下的 23 位是规一化数（或数的科学表示法形式）的小数部分（见表 2.3.1）。

表 2.3.1		IEEE 32 位浮点数
符 号 位	指 数 部 分	小数部分（尾数）
S	EEEEEEEE	FFFFFFFFFFFFFFFFFFFFFFF

对于 32 位浮点数来说，精度等于其尾数最低位的位值或粒度的一半。由于第一位的位值是 2^E （指数偏差可忽略不计）并有 23 位的小数，LBS 的位值就是 2^{E-24} 。例如，非常接近 1 的数，粒度就是 2^{-24} ，或大约是 0.0000001192。接近 1000 的值，粒度增加到 $0.000061 (2^{-13})$ ，接近 100 000 的值，粒度就是 $0.0078125 (2^{-7})$ 。用实际的术语来讲：以米作为单位，我们的世界是边长为 100 公里的正方形，在我们世界的最远角落，32 位的浮点数仅能让我们来表示粒度为 7.8 毫米的空间。我们要表示的坐标系越大，在最远的范围内精度就越低。要表示相当于美国本土大小（从东到西约 4500 公里）的区域，在离原始出发点最远的海岸，你将受限于半米的粒

度（见图 2.3.1）。

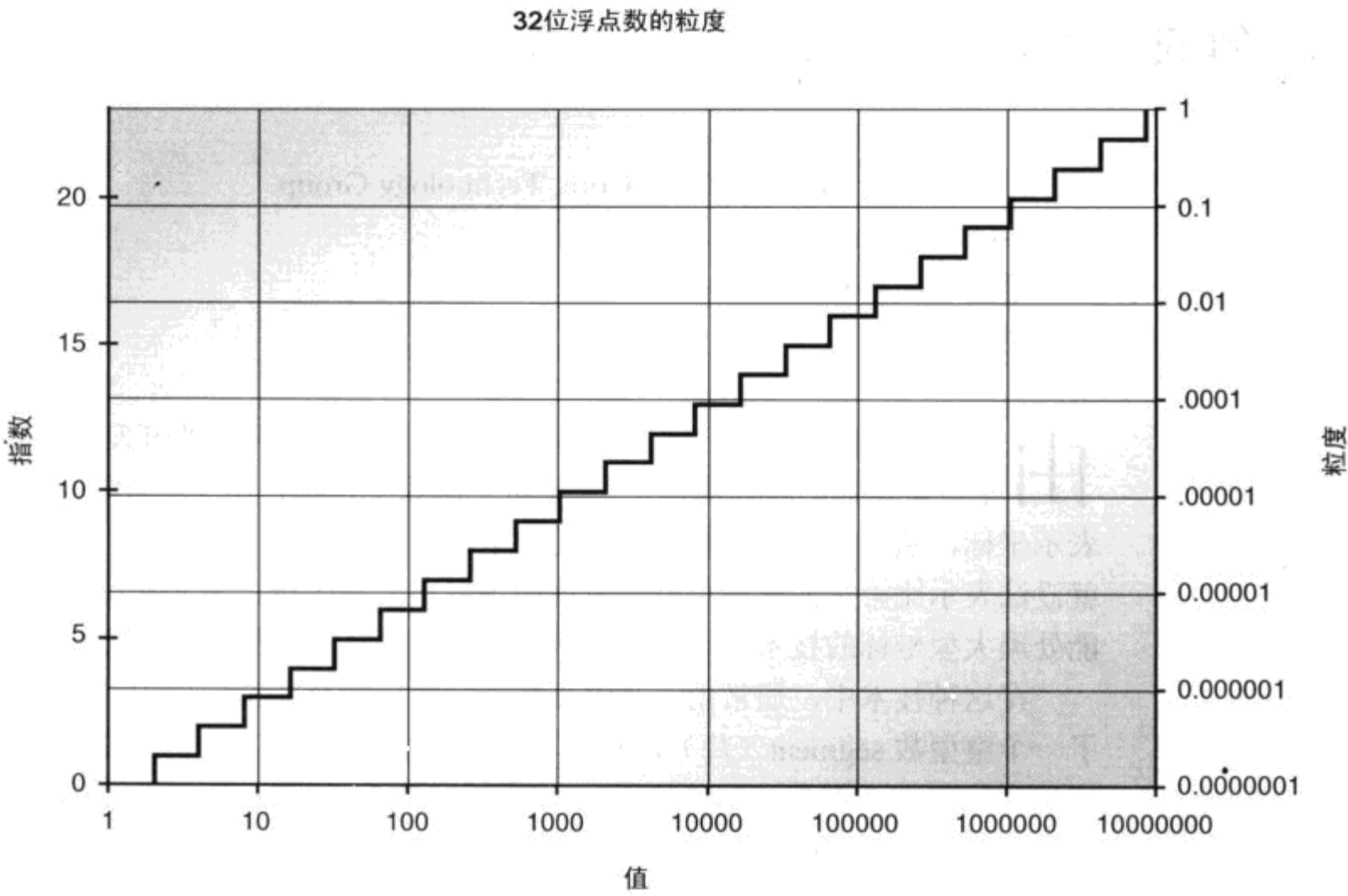


图 2.3.1 浮点数的粒度

精度误差出现的征兆

这是不是意味着如果你只是创建几公里见方的户外区域，就不需要担心精度问题？可惜的是，就算是有些数字，表面上看在能提供超过足够精度的范围内，但精度误差的征兆也会出现。这个问题的主要原因在于我们最常用的不是简单地用数字来描述位置，而是将它们用于计算，当中误差可能会放大。像模型骨骼之类的分层对象图，通常靠一系列的矩阵乘法来决定在照相空间中的位置和网格方向。当连接几个矩阵乘法后，通常不起眼的精度误差可能已经增加到会引起视觉不连续，或改变物理模拟的行为的地步。

由于用一套数学计算来精确得到每一步引起的误差数量是可能的，因此可以判定引起的最大误差是否值得去解决，然而这是一个让人头疼的费力过程。如果当误差很明显时，只简单地处理精度问题要直接了当得多。

- **无法将相邻的对象对齐。**这是最常见的误差的征兆，在这情况中，由于数字粒度的原因，你实际上不能在任意理想的位置或方向上设置对象。这最常发生在奇数角度的对象上。
- **网格中的裂缝。**这是第一个问题中的变种，但是在这个问题中，就算是很小的误差也是可见的。邻接的地形网格可能在本来互相吻合的地方显示出裂缝，通过缝隙露出天空或填色。由多个网格模型组合而成的角色模型可能就无法完全封闭，暴露出很难看的裂缝。
- **动画抖动。**有骨骼动画（skeletal animation）的模型可能看起来会颤抖或摇动；当动

在我们继续说明本文的核心解决方案之前，还有一些值得一提的处理精度问题的方法。在某些情况下，它们可能是代码影响最小的最佳选择。

1. 使用高精度浮点类型

精度解决的最理想方式就是简单地采用较大浮点类型。IEEE 双精度 64 位浮点数使用了 52 的尾数，超过单精度浮点数 23 位的两倍。多出的 29 位则提高了相对精度，相当于 536 870 912 倍。切换到双精度是否可行完全取决于你的平台。当代消费芯片能够在单精度和双精度模式下做浮点运算。然而，在较低精度中的运算有很明显的性能好处，这就是为什么大多数游戏会切换 FPU (Float Point Unit, 浮点处理单元) 状态到单精度模式，在运行期间不变。单精度浮点数也只要双精度存储容量的一半，这对细节极多的几何图形而言意义很大。最后，大多数图形 API 不处理单精度浮点数，图形硬件可能在更低精度中做内部操作。即使你能够成功地将你的引擎转换到使用双精度浮点数，只要图形硬件还用较低精度做转换，精度问题仍会突然出现。

2. 消除全局空间变换

大多数图形流水线包含了应用于顶点的变换序列，如下：

$$M_{\text{world}} \rightarrow M_{\text{view}} \rightarrow M_{\text{projection}}$$

全局变换 (M_{world}) 从模型空间改变坐标，其中顶点是相对于模型的本地原址而定义的，对于全局空间来说，顶点是相对于同一场景中所有对象的原点而定义的。基本上，全局变换将模型放入全局中，也就是它的名字。视图变换 (M_{view}) 在全局空间中定位观察者，将顶点变换到照相空间。视图矩阵在 camera 位置周围重新定位游戏世界中的对象(照相空间的原址)和方向。在大型坐标中， M_{world} 和 M_{view} 都可能有精度误差。

大多数引擎会将 M_{world} 、 M_{view} 和 $M_{\text{projection}}$ 单独地递交给它们的渲染流水线，或是 M_{world} 和 $M_{\text{viewprojection}}$ 的一个连接。将 M_{world} 和 M_{view} 尽可能早地相连，然而，随之递交 $M_{\text{worldview}}$ 和 $M_{\text{projection}}$ 的话，也就跳过了许多精度问题。这特别适用于解决动画的抖动问题。

例如，在骨骼层次中，顶点经历了以下变换：

$$[M_{\text{world}} \times M_{\text{parentbone}} \times \cdots \times M_{\text{rootbone}}] \rightarrow M_{\text{uiew}} \rightarrow M_{\text{projection}}$$

通常，所有的骨骼变换都是从左到右连接，层次向上，产生了将顶点变换到全局空间的一套矩阵，其中，它们能被混合、照亮等等。如果 root 骨骼矩阵 M_{rootbone} 在连接到 child 骨骼变换之前和视图矩阵 M_{view} 相结合，那么渲染精度误差就能够被隔离到一个单独的操作中，和根骨骼和视图矩阵相结合。

$$[M_{\text{bone}} \times M_{\text{parentbone}} \times \cdots \times (M_{\text{rootbone}} \times M_{\text{view}})] \rightarrow M_{\text{projection}}$$

对于需要递交离散视图矩阵的流水线、几何图形可以和一个标示视图矩阵一起递交。然后所有顶点的混合、照亮和原始计算就在视图空间中完成，没有产生额外的精度误差。

虽然这可以减轻一些在动画中比较明显的视觉问题，也可以作为解决精度问题的保留招数，但它无法解决基本问题，简单来讲，就是没有足够的精度来表示游戏世界中的位置。此外，只要视图矩阵改变——基本上就是每次 camera 移动的时候，它就会在每个网格引入额外的 CPU 级的转换。

2.3.3 偏移位置



本文给出的解决方案注意到了之前罗列的所有需求，也基本上包括了介于定点和浮点数之间的混合体。通常由三个浮点数组成的位置矢量，增加了代表浮点原点的额外整型数组件。由于该数据表达的方式类似于分段地址扩展早期 32 位处理器地址空间，组件用 `segment` 和 `offset` 来命名，其结合体叫做偏移位置。这里所罗列的类和数据结构的完整代码附在配套光盘中。为了简单起见，在文字中用了简化的定义。

```
class FarPosition
{
private:
    FarSegment    m_segment;
    Vector3       m_offset;
    static float  s_segmentSize;
};
```

空间中的所有点都是相对于某种概念上的固定点（我们称之为原点）来度量的。由于浮点数的粒度是数量级相关的，离原点越远，粒度越大。重新定位原点，这样可以使它靠近我们所要度量的点，这样我们就能减少我们度量的粒度。偏移位置让原点能够动态定位，使得有效精度最大化。偏移位置的片（`segment`）部分代表了 2D 空间中的原点，假设游戏世界模型中高度是用 y 轴表示，那么它在这个例子中就是 xz 平面。由于在外面渲染环境中的精度误差产生于水平位置，而不是高度的数量级，所以做出这个选择来将片限制到 xz 平面。

图 2.3.2 说明了一个特殊的片如何相对于游戏世界原点而存在。片空间由所有相对于片原点的点组成。由于我们关注于充分利用有效的精度，所以我们对位置作规一化，这样，规一化后的片空间由那些偏移离开片原点不超过片大小的位置组成（用灰色表示）。

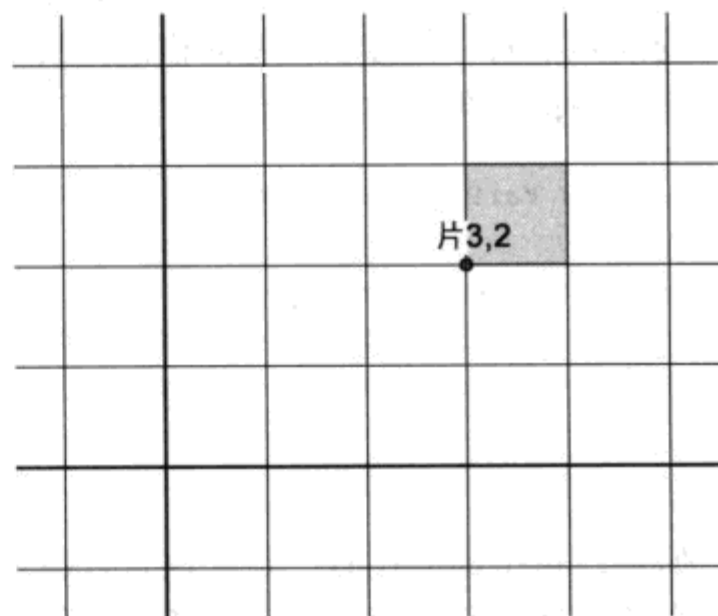


图 2.3.2 片的布局

1. 选择片的大小

片大小的单位选择有些随意性。由于将这编码到全局常量中，它就能在我们所想得到的精度和最大可表示范围及重新规一化损耗之间获得平衡。当重新规一化偏移后，它们就受限于至多一个片大小的范围；这就仿佛在允许的最大精度误差上多了个夹子。1024.0f 的值为以米为选择的，为基于户外环境的典型人提供了合理的范围。单位或比例的不同大小会影响片大小的选择。由于片值用以整数单位为偏移来表达原点，为片大小选择 1024.0f 则给出了相

当大的工作范围。我们可以用 16 位带符号的值来表达片，那么片的最大表示范围则为 $\pm 33553408.0f$ ($32767 \times 1024.0f$)；这足以表达整个地球表面的平面投影。将两个 16 位的片坐标打包到一个 32 位的值，**FarPosition** 结构的大小（其中包含三个表示偏移的浮点数）总共 16 字节，正是个结构对齐和打包的合适尺度。下面给出 **FarSegments** 的定义：

```
union FarSegment
{
    struct
    {
        int16 x, z;
    };
    int32 xz;
    matFarSegment() {};
    matFarSegment( ZeroType ) : xz(0) {};
};
```

使用联合（union），我们就可以将 **FarSegment** 视作一个 32 位的值，这样大大简化和加快了初始化和对比。**ZeroType** 初始化器（initializer）为初始化到一个已知状态提供了基于类型的重载，使得我们能够简单地提供显示的初始化构造器和空构造器（在标准容器类中储存是相当有用的）。

2. 重新规一化的细节

带偏移位置的全局位置的表示已经充分直接明了，但我们需要将传统位置编码成偏移位置和变换回来的一种方法。将偏移位置转换回传统矢量表示相当清除；我们只需要根据片的大小增加偏移和合适比例的片。

```
Vector3 FarPosition::GetApproximateVector() const
{
    Vector3 v = m_offset;
    v.x += m_segment.x * s_segmentSize;
    v.z += m_segment.z * s_segmentSize;
    return v;
}
```

但是，对于该简化有一点要注意。执行该操作后的结果落在标准浮点数表示的范围内，这意味着结果的精度减少了。变换操作应该最简化，只有当需要的时候才执行。这是根据成员函数的名字 **GetApproximateVector()** 而得到的。

从传统表示转换到偏移距离，也不是难事。

```
void SetFromVector( const Vector3& vector )
{
    m_segment.xz = 0;
    m_offset = vector;
    Normalize();
}
```

当我们最初存储矢量的时候，偏移位置处于未经规一化的状态。偏移的数量级是未知的，

可能远大于我们所认定的合适的片大小。因此，我们在偏移位置上调用一个规一化过程。

```
void FarPosition::Normalize()
{
    if ( fabsf(m_offset.x) >= s_segmentSize )
    {
        m_segment.x += FloatToInt(m_offset.x / s_segmentSize);
        m_offset.x = fmodf(m_offset.x, s_segmentSize);
    }

    if ( fabsf(m_offset.z) >= s_segmentSize )
    {
        m_segment.z += FloatToInt(m_offset.z / s_segmentSize);
        m_offset.z = fmodf(m_offset.z, s_segmentSize);
    }
}
```

规一化后，偏移 x 和 z 部分回落在范围 $(-s_segmentSize, +s_segmentSize)$ 内。要注意的是，过程包括了一个浮点到整型的转换，所以你应该用标准优化技巧来加快转换。

当从 **Vector3** 表示转换到偏移位置时，结果将只和最初的 **Vector3** 一样精确。为了利用已经提高的有效精度，我们需要为所有绝对位置表示使用偏移位置，而不是仅仅在渲染的时候。这意味着，对象位置应该始终用偏移位置来表示，编辑者应该用偏移位置来操纵对象。任何时候我们将绝对位置转换到 3 个浮点矢量，就会丢失精度。

由于规一化是独立基于偏移的数量级，所以可以将它和可能产生非规一化偏移的偏移位置上的操作相结合。

```
void FarPosition::Translate( const Vector3& vector )
{
    m_offset += vector;
    Normalize();
}
```

偏移位置的关键行为之一就是为它们为特定的片提供偏移的能力。方法如下所示：

```
Vector3 FarPosition::GetRelativeVector( const FarSegment& segment )
const
{
    Vector3 r = m_offset;
    r.x += (m_segment.x - segment.x) * s_segmentSize;
    r.z += (m_segment.z - segment.z) * s_segmentSize;
    return r;
}
```

当我们讨论偏移位置是如何和渲染流水线相配套的时候，对此的需要就会变得比较明确。我们可以用增加偏移和片（比例合适）间的距离，来度量两个偏移距离间的距离。

```
Vector3 operator-( const FarPosition &lhs, const FarPosition &rhs )
{
    matVector3 r = lhs_offset - rhs.m_offset;
```

```

    r.x += (lhs.m_segment.x - rhs.m_segment.x) * s_segmentSize;
    r.z += (lhs.m_segment.z - rhs.m_segment.z) * s_segmentSize;
    return r;
}

```

如果我们要得到在两个偏移之间的标量距离（scalar distance），那么仅需要得到返回的矢量的标量长度就可以了。

要说明的是，执行这步操作分两步走，首先将两个偏移位置变换到全局空间的 `Vector3` 坐标，接下来做标准的矢量减法。然而，这样做可能会引起精度丢失，如果位置远离原点，当你考虑将会发生什么，那就会一目了然。

如果点之间的距离很大，一个 `Vector3` 可能还无法在没有精度误差的情况下表示距离。由于我们很少关注间隔很远的对象的相交，所以这是可以接受的。如果我们需要简单地测定两者之间的距离，有最大的相对而非绝对误差的结果通常已经够了。

2.3.4 渲染流水线变化

现在我们有了用大大提高后的精度来表示位置的方法，但是，如何在渲染流水线中利用它呢？渲染引擎处理的是矩阵变换，而不是位置矢量。那么我们应如何创建矩阵，使得精度误差最低呢？答案相当直接：我们在相同的片空间（segment space）生成所有的矩阵。虽然，在细节中会有麻烦。

首先，让我们来研究下通常是如何表示位置和坐标的。方法之一，也是个很自然的方式就是用一个 `Vector3` 来表示位置，再用一个四元数来表示方向。

```

class Transform
{
private:
    Quaternion      m_quaternion;
    Vector3         m_position;

    mutable Matrix4x4 m_matrix;
};

```

这样一个类可能也足以马马虎虎生成一个所需的 4×4 矩阵，并将结果缓存进对象中。我们可以将这个类包装到一个新的能够用偏移位置来表示变换的类。

```

class FarTransform
{
public:
    void SetBasisSegment( FarSegment segment );
    const matTransform& GetLocalTransform() const;
    mutable bool      m_bPositionDirty;
    mutable FarSegment m_basisSegment;
};

```

这个类将一个本地变换和一个偏移位置和一个基片（basis segment）相结合。这个基片是本地变换所在的片空间。在相同的基片或片空间中生成所有的渲染变换，所有的矩阵，这

样渲染就可以同往常一样继续。要这样的话，在场景图中的所有根对象，包括 camera，都需要使用 FarTransforms。而诸如独立骨骼转换的非根对象，就可以继续使用标准的转换表达，这是因为它们相对于父母的偏移通常是最小的。这也避免了除需要情况之外的任何偏移位置的性能损耗。

1. 方便地对基进行重规一化

我们应该使用什么样的基片？答案在某种程度上取决于我们的使用模式。如果我们正在渲染的话，选择基于照相位置的基片就很自然。如果 camera 在运动中（它是持续地），那么只有当 camera 移动的距离超过片大小时，来自规一化后的照相位置的片就会改变。因为允许偏移能从片原点衍生可正可负的距离，就避免了当照相运动处于非正常情况下，无谓地改变片。如果我们不关注渲染，而注意在一空间区域中很多对象间的相交（例如，处理在连续大型空间中以区域的服务器），那么在所感兴趣的区域中心，选择固定基片会更有意义些。区域中的所有对象能够在本地的转换上操作，这样可以使精度误差最小化。

当基片改变会发生什么？第一要注意的是，必须确保在渲染的任何对象都有相对于当前基片的本地变换。就和我们为 FarTransform 所做的一样，这可以用记录为对象所使用的最后基片来做到。设置一个和当前不同的基片能让本地变换失效。

```
void FarTransform::SetBasisSegment( FarSegment segment )
{
    if ( segment != m_basisSegment )
    {
        m_bPositionDirty = true;
        m_basisSegment = segment;
    }
}
```

当我们接下来请求本地变换时，检查 dirty 标志位，如果该位有效的话，说明相对于新的基片，本地变换已经重定位了。

```
const Transform& FarTransform::GetLocalTransform() const
{
    if ( m_bPositionDirty )
    {
        // 存储位置
        m_localTransform.SetPosition(
            m_position.GetRelativeVector(m_basisSegment));
        m_bPositionDirty = false;
    }
    return m_localTransform;
}
```

2. 全局空间与本地空间

和偏移位置相结合的最大障碍之一就是，除了在特别处理和偏移位置相合并的个体之外，我们无法以任何有意义的方式来说明全局空间；所有其他位置的表示都相对于当前的基

片。只有两个本地变换在相同基片中，才能合并它们之上的操作。如果有一个 4×4 的矩阵或一个三浮点的矢量，就知道我们有个本地（相对于基而言）位置或变换，但如果不检查包含它的对象，那么就不知道它相对于哪个基。所有东西都是相对的，这个观点会引起混淆，也是引起 bug 的可能原因。

在计算的时候，我们也需要区别对待相对位置信息和绝对位置。我们用偏移位置来表示绝对位置，用 `Vector3` 来表达相对位置（位置）。这个定义抛弃了我们所受限的特定操作。例如，我们不需要有个操作符来相加两个偏移位置。除了结果域没有意义外，提供这样的操作符还给我们带来了片溢出的危险。可以研究一下后果，打个比方，我们选择为几百个对象独立地计算片坐标和偏移的综合。这可能需要找到对象的平均位置。片的值很容易就会溢出有效范围。处理该问题的正确途径就是要，选择一个合计所有相对偏移所在的片。

```
typedef vector<FarPosition> VFP;
FarPosition FindAveragePosition( const VFP &positions )
{
    if ( positions.empty() )
        return FarPosition(Zero);

    FarSegment segment = positions.front().GetSegment();
    Vector3 offset(Zero);
    for ( VFP::iterator it = positions.begin(); it != positions.end(); ++it )
    {
        offset += it->GetRelativeVector(segment);
    }

    FarPosition average;
    average.SetSegment(segment);
    average.SetOffset(offset / positions.size());
    average.Normalize();
    return average;
}
```

该函数使用来自矢量的第一个位置作为引用参数。它然后合计在那片空间每个位置的相对位置。就算有成千上百个位置，只要它们还算是本地的，那结果就应该是精确的。有无数远程对象的非正常情况也会工作正常，没有溢出，但会受限于传统偏移的相同精度限制。由于这些偏远位置间的相交的需要几乎不存在，所以这不是个很明显的局限。

3. 变换到本地空间

通常，我们将某些建筑保存在绝对地形空间里，但我们需要将它变换到非根场景图节点的本地空间。在两种情况下需要做以上变换：碰撞相交测试，或选取射线和场景的相交测试。在选取射线的情况中，我们始于以偏移位置表示的射线原点，通常以 `Vector3` 表示射线方向，以标量的最大选取距离。将这变换到以 `FarTransform` 表示的本地对象空间由两步组成：变换射线原点，变换射线方向。

变换射线方向相当直接。

```
// inputs:
```

```
// Vector3 vDir = world direction
// FarTransform transform
Transform& localTransform = transform.GetLocalTransform();
Matrix4x4& invMatrix = localTransform.GetInverseMatrix();
Vector3 localDir = invMatrix.TransformVector(vDir);
```

这等同于如果不使用偏移位置我们所用的过程。变换射线原点相对于通常过程，只有个不大的改动。

```
// inputs:
// FarPosition vOrg
Vector3 relOrg = vOrg.GetRelativeVector(transform.GetBasisSegment());
Vector3 localOrg = invMatrix.TransformPoint(relOrg);
```

在用矩阵逆转来变换点之前，我们找到了相对于本地变换的基片的位置。至于基片是什么则无关紧要；关键在于它匹配于本地变换所存在的相对空间。如果我们注重于在计算中保持最高的精度，那么我们可以在操作开始时，特别将基片设成来自于 FarTransform 中被规一化的 FarPosition 所在的片。这对于和渲染无关的系统可能比较重要，因为它们不依赖于一个已经为 FarTransform 所设置的相应基片。

2.3.5 对性能的思考

在实现偏移位置中有一些潜在的缺陷。如之前提到的，为了避免片移 (segment-shift) bug，保持对所有变换的相对特性的认知是基本的。这些都可能有问题，因为在简单测试实例中看起来正常工作的代码，当对象、camera，还有基片都在改变的时候，可能会错误百出。

使用偏移位置带来的性能影响通常可以忽略不计，有以下几个理由。

- 规一化测试只需要在动态根节点对象上完成，包含简单的浮点对比。
- 对于渲染而言，从最后所使用的基片原点出发，沿着 x 轴或 z 轴，基重新规一化只有当 camera 比片大小要远的时候才会发生。
- 简单更新根变换位置，就可以完成基的重新规一化。该计算只包括一些整型和浮点数的操作。

当用多个 camera 同时观察全局区域时，可能会发生这样的问题。如果 camera 足够远，那么它们将看见完全不同的对象集合，整个系统没有问题。然而，如果 camera 只是快到看见相同对象的地步，可来自照相位置的片是不同的，那么当渲染每个视图时，它就会更新对象变换中的基片，两个视图就会在要用的基片上打架，而且基的重新规一化会在每次渲染的时候发生。例如一部 camera 用作用户的视口，而另一部 camera 用来为场景里的对象渲染环境地图时就可能发生这个问题。这个难题的解决方案是选择一个实体，由其决定用于渲染的基片。这可以是主 camera，也可以根据玩家的位置来决定。

2.3.6 结论

创建大尺度游戏世界的开发者必须应付浮点精度相关的问题。偏移位置给浮点精度问题

提供了一个解决途径。它速度快、硬件兼容性良好、可调，且不需要较高的精确浮点数或是浮点技巧。它需要在全局空间上考虑基本的移位操作。

2.3.7 参考文献

[Goldberg91] Goldberg, David, "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, March 1991.

[Wilkinson94] Wilkinson, James H., *Rounding Errors in Algebraic Processes*, Dover Publications, 1994.



2.4 非均匀样条

作者: Thomas Lowe, Krome Studios

E-mail: tomlowe@kromestudios.com

译者: 许竹钧

审校: 沙鹰

样条 (spline) 是弯曲的路径, 通常定义为一系列点, 这些点在 3D 空间被称为节点 (node)。有时候每个节点需要更多的信息来描述曲线的形状。通常用一个像 `GetPosition(float time)` 的简单函数来评价样条, 其中, `time` 参数的取值范围是 $0 \sim 1$ 。

本文描述了三种类型的非均匀三次样条。非均匀样条有个有用的特点, 那就是它们的速率不受节点间距离的影响, 这使得它们在游戏开发中特别有用。这三种样条类型如下。

- **圆形非均匀样条:** 近似不变的速率, 对于沿轨道行驶的列车有用。
- **平滑非均匀样条:** 2 阶连续或连续加速, 对特效粒子的路径有用。
- **时控的非均匀样条:** 之前样条的变种, 具有可指定的时间间隔, 例如, 用于分镜头 camera。

2.4.1 样条的种类

以下是常用的样条类型列表, 图 2.4.1 给出了基本类型的比较 (更多信息参考 [Demidov03])。 n 阶连续的意思是, 样条的 n 阶导数连续。

- **贝塞尔 (Bezier) 曲线:** 这些样条只经过起点和终点, 所有导数都是连续的。不像其他类型的样条, 每增加额外的节点, 它们的复杂度就会增加。
- **Catmull-Rom 样条:** 经过每个节点。2 阶连续。
- **Kochanek-Bartels 样条:** Catmull-Rom 的扩展, 每个节点需额外的参数。
- **自然三次样条:** 通过每个节点。2 阶连续。
- **立方 B 样条:** 不通过每个节点。2 阶连续。
- **NURBS:** 立方 B 样条的扩展, 每个节点需额外的参数。能够定义确切的圆、双曲线和椭圆。

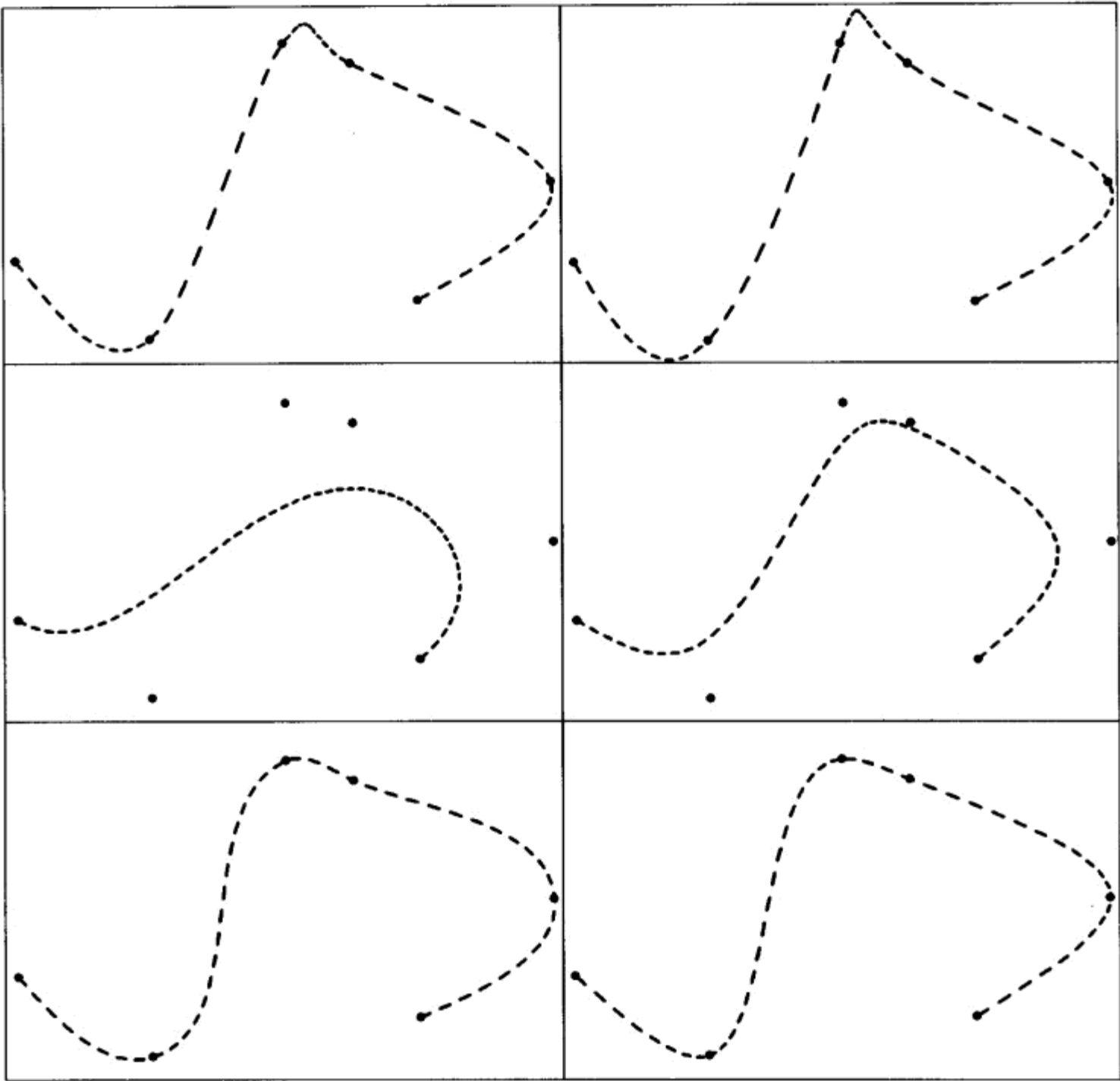


图 2.4.1 样条的类型，从左到右、自上而下依次是：Catmull-Rom 样条、自然样条、贝塞尔样条、B 样条、圆形非均匀样条和平滑非均匀样条

2.4.2 三次样条的基础理论

本文中描述的非均匀样条非常接近于 Catmull-Rom 和自然三次样条。它们都经过每个节点（给出高级控制的有用属性）它们都是分段的；两节点间的曲线部分是一个时间的独立函数。最后，它们都是三次的，意味着函数是以 $p(t) = at^3 + bt^2 + ct + d$ 的形式表现，其中 $p(t)$ 是在时间 t 的位置。

三维样条的每一维都是三次的，所以我们可以（用矢量符号）这样写：

$$\begin{aligned} p(t) &= at^3 + bt^2 + ct + d \text{ 或} \\ p(t) &= [t^3 \quad t^2 \quad t \quad 1]A \quad (A \text{ 为矩阵}) \end{aligned} \tag{2.4.1}$$

来看一个单独的段，对该曲线的完全描述可以用起点和终点位置，还有起始和终止速率（或用切线）（见图 2.4.2）。

在该段中，如果我们让 t 在 $0 \sim 1$ 之间伸缩，然后经过一个时间单元，如果节点速率矢量继续在一条直线上，我们可以把它描述为位置中的变化。

$A = HG$ 所以

$p(t) = tHG$ 其中

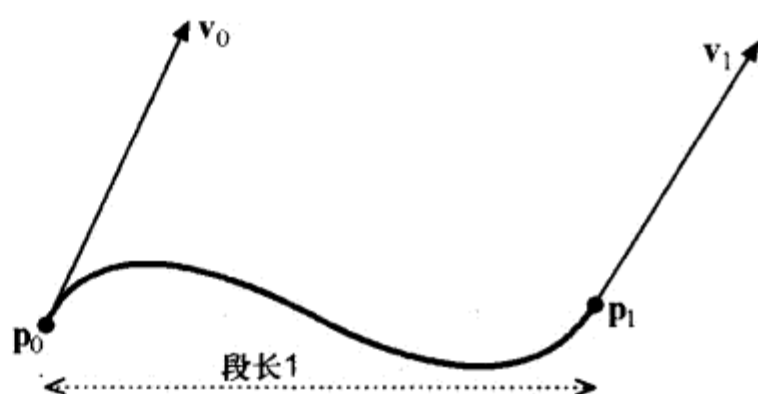


图 2.4.2 用四个矢量确定的一段三次曲线

$$t = [t^3 \quad t^2 \quad t \quad 1] \quad H = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad G = \begin{bmatrix} P_0 \\ P_1 \\ v_0 \\ v_1 \end{bmatrix} \quad (2.4.2)$$



这里， t 是时间矢量， H 是 Hermite 插值矩阵， G 是几何矩阵。Hermite 插值矩阵是惟一满足我们描述的矩阵。它的推导见[Hermite99]。所以现在我们可以得到在任何时间 $0 \leq t \leq 1$ 这段曲线上的位置。公式 2.4.2 已经由在配套光盘中的函数实现了。

Catmull-Rom 样条和三次样条都是为这些三次段而设计的，只是用不同的方法来选择节点的速率矢量。由于它们是均匀的，它们的节点沿着时间线是等距离分布的。在某个时间 ($0 \sim 1$)，沿着样条返回一个位置，只要找到相应的段和它的 t 值，然后再应用公式 2.4.2 就行了。

2.4.3 圆形的非均匀样条

图 2.4.3 给出了一条赛道，是由左边的 Catmull-Rom 样条和右边的圆形非均匀样条创建的。注意非均匀分布的节点是如何严重扭曲了速率和左边样条的形状，反之，右边的样条有着极为规则的速率和形状。

因此，要描述空间曲线（如列车轨道和测量几何），或要在不变速率穿越的路径（如敌方巡逻路径或者是铁路上的对象等），圆形非均匀样条（RNS）是非常有用的。此外，在需要圆形的拐角的地方可以使用这些样条，可以在外面控制穿越速率。图 2.4.3 中的赛道就是一个很好的实例。

实现

如何实现 RNS？首先，将样条的时间线根据段长（这是非均匀性部分）按比例分割。可以用这里节点间的线性距离作为段长的近似值。分割意味着均匀穿越样条的对象将在每对节点间以相同均速穿越。以下伪码为一个简单的数据结构和寻找函数。

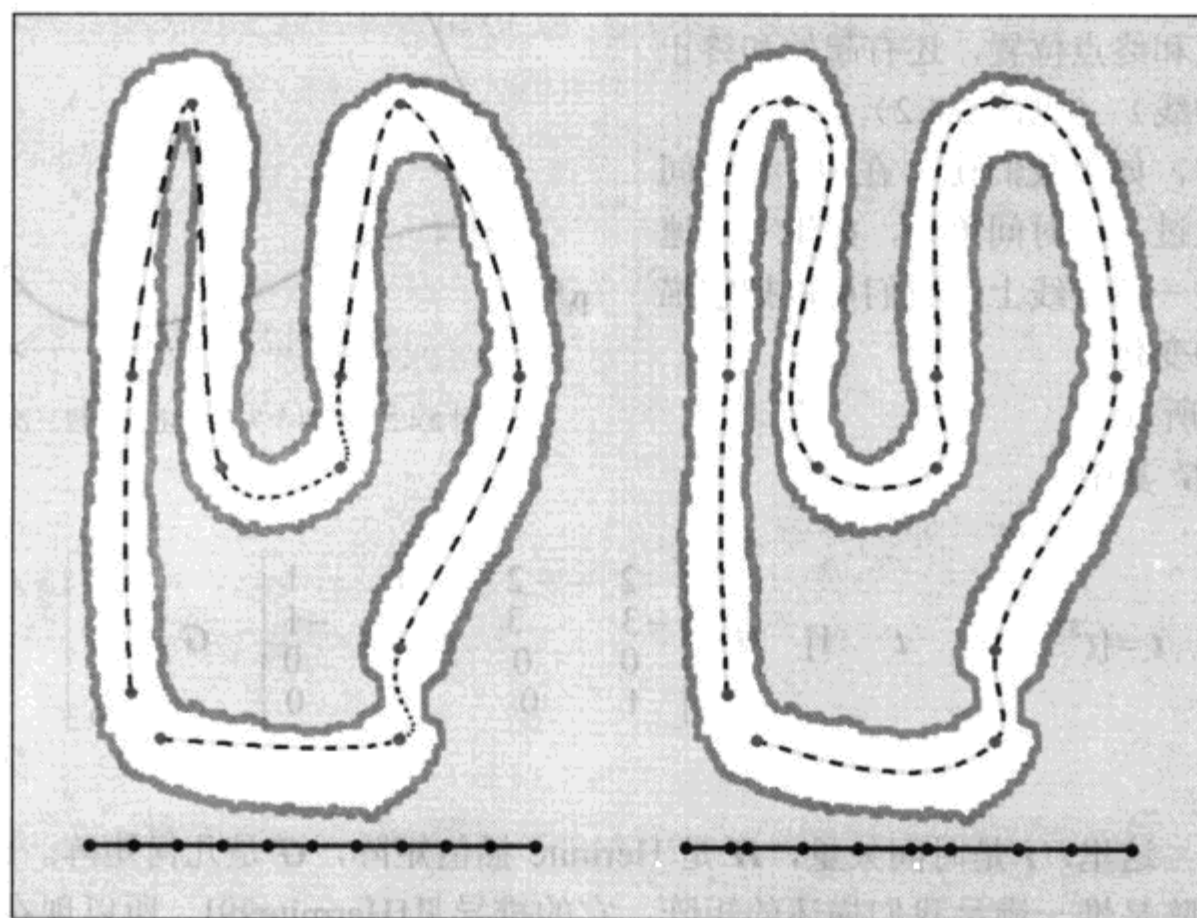


图 2.4.3 Catmull-Rom 样条（左）和圆形非均匀样条（右）的比较。

在每条道下方，我们会看到在时间线上分布的节点

```
struct Node {
    Vector position, velocity;
    float distance; // 和数组中下一节点之间的距离
} node[10];

Vector GetPosition(float time)
{
    float distance = time * maxDistance;
    float currentDistance = 0.f;
    int i = 0;
    while (currentDistance + node[i].distance <
           distance && i < 10)
    {
        currentDistance += node[i].distance;
        i++;
    }
    float t = distance - currentDistance;
    // i 是段号, t 是沿着段的时间

    // 备注: 函数 a 余下部分将在下面给出
}
```

现在每一段都是定义在不同时间域上的三次函数。如果我们能重新调整该三次函数的定义域，让它在 $0 \leq t \leq 1$ 范围工作，那么我们就继续能在每一段上使用公式 2.4.2。这意味着用调整节点速率 v^N 的方法来得到那段曲线的时间起点和终点速率 v^S 。首先时间转换：

$$\Delta t^s = \Delta t^w * \Omega / l \quad (2.4.3)$$

其中 Ω = 沿着样条的穿越速度
= 样条长度 / 穿越样条的时间

通俗地讲, 调节那段曲线 t 的值 (0~1) 是用样条被穿越的速度再乘上绝对时间, 并除以那一段的长度而得来的。可以看到长为 10m 的段穿越 ($\Delta t^s=1$) 需要的时间长为 1m 的段的 10 倍。

由公式 2.4.3, 我们得到

$$v^s = v^w * l / \Omega \quad (2.4.4)$$

假设穿越速度为单位速率的话, 该节点速率就等于绝对速率。

$$\text{所以 } v^s = v^N * l \quad (2.4.5)$$

换句话说, 传到 `GetPositionOnCubic()` 的速率必须首先乘以该段的长度, 这样就可落在正确的时间区间上。我们用添加的方式来完成寻找函数。

实现 RNS 的第二部分是获取节点速率 (这指的是 RNS 的圆形部分)。我们选择它们作为单位长度 (因此, 节点所在的速率等于节点之间的平均速率), 并且将之前一个和下一个节点间的角度平分 (见图 2.4.4)。

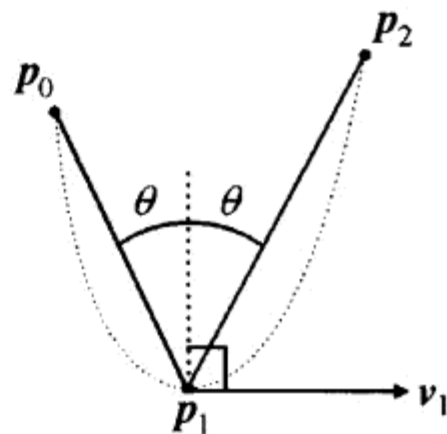


图 2.4.4 节点速率矢量的选择



创建和访问 RNS 的完整代码附在配套光盘中。

```
t /= node[i].distance; // 把 t 缩放在范围 0~1
Vector startVel = node[i].velocity * node[i].distance;
Vector endVel = node[i+1].velocity * node[i].distance;
return GetPositionOnCubic(node[i].position, startVel,
                          node[i+1].position, endVel, t);
```

2.4.4 平滑非均匀样条

图 2.4.5 将一条平滑非均匀样条 (SNS) 和一条 RNS 还有自然三次样条 (平滑均匀样条) 相比。要注意, 为什么 RNS 速率不受节点间隔影响, 但是自然样条的运动显得很平滑。

SNS 是二阶连续的, 也就是说, 穿越它的时候加速度是连续的, 这点和一阶连续的 RNS 有所不同。它是一条平滑的时间曲线, 所以要描述加减速流畅, 并且没有回转圆 (turning circle) 的物体的运动, 是非常理想而又合适的。例如, Camera 的移动、飞碟的路径、手写体逼近 (approximating handwriting), 或创建特效粒子轨迹。值得一提的是, 3ds max 中的“点曲线”就是采用的该样条。

实现

SNS 和 RNS 的工作方式一致; 只不过节点的速率是不同的, 使得 SNS 是二阶连续。在

这里，复杂性主要源于，必须基于节点位置来生成这些节点速率。

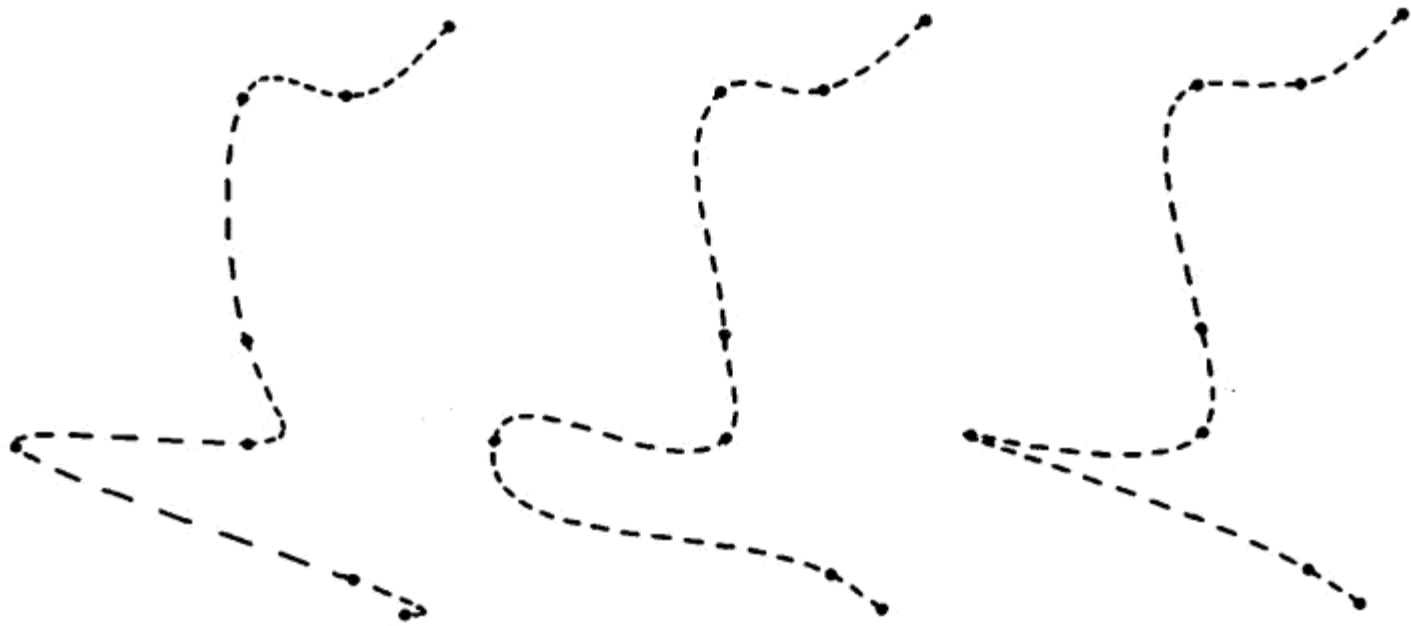


图 2.4.5 自然样条、RNS 和 SNS

要使得该样条二阶连续，必须要为每个节点选择速率矢量，这样前一段曲线的最终加速度等于后一段的曲线起始加速度。

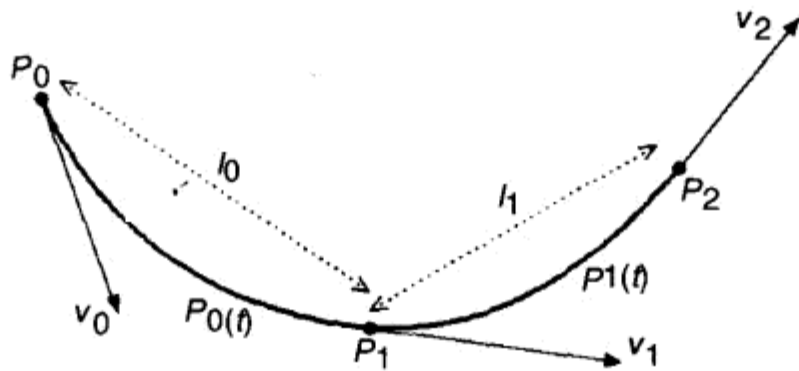


图 2.4.6 当 $a\theta^w(l) = aI^w(0)$
SNS 是二阶连续的

(2.4.6)

这样的话，什么是加速度的公式？在一段上，对公式 2.4.2 求导，推出：

$$p(t) = (2p_0 - 2p_1 + v_0^s + v_1^s)t^3 + (-3p_0 + 3p_1 - 2v_0^s - v_1^s)t^2 + v_0^s t + p_0 \tag{2.4.7}$$

因而
$$v^s(t) = 3(2p_0 - 2p_1 + v_0^s + v_1^s)t^2 + 2(-3p_0 + 3p_1 - 2v_0^s - v_1^s)t + v_0^s \tag{2.4.8}$$

$$a^s(t) = 6(2p_0 - 2p_1 + v_0^s + v_1^s)t + 2(-3p_0 + 3p_1 - 2v_0^s - v_1^s) \tag{2.4.9}$$

然而，我们要求绝对加速度。我们可以从公式 2.4.3 推导出：

$$a^w = a^s * \Omega / l^2 \tag{2.4.10}$$

转换公式 2.4.9，再应用到公式 2.4.6，我们就得到：

$$\frac{6(2p_0 - 2p_1 + v_0^S + v_1^S) + 2(-3p_0 + 3p_1 - 2v_0^S - v_1^S)}{l_0^2} = \frac{2(-3p_1 + 3p_2 - 2v_1^S - v_2^S)}{l_1^2} \quad (2.4.11)$$

然后, 应用公式 2.4.5, 解出 v_1 , 我们得到:

$$v_1^N = \frac{l_1(3(p_1 - p_0)/l_0 - v_0^N) + l_0(3(p_2 - p_1)/l_1 - v_2^N)}{2(l_0 + l_1)} \quad (2.4.12)$$

现在我们得到了用相邻两个节点的速率来表示的节点速率。假设节点速率可以赋值给第一和最后一个节点(稍后定义), 有两种方法可以用来解这个系统。

第一个方法, 公式 2.4.12 组成了公式的三对角系统, 它的时间复杂度为 $O(n)$ 。关于三对角系统的解法请参考[RecipesC93]。



第二个方法, 也就是在示范代码中所实现的方法, 它反复应用公式 2.4.12 于每个节点。可以将其考虑为一个适用于样条的平滑过滤器, 它减少了每次计算的加速度不连续性。在实际情况中, 4 次计算中, 仅有 3 次需要得出尽可能精确的解。具体细节请参见配套光盘中的函数 Smooth()。

现在我们得到了这个过滤器, 一条 SNS 简单来说就是在所有节点添加后, 于其上调用了数次 Smooth() 的 RNS。



图 2.4.7 SNS 实例

2.4.5 时控的非均匀样条

时控的非均匀样条 (Timed Nonuniform Spline, TNS) 是 SNS 的一个扩展, 每个节点带有额外的参数: 该节点到下一个节点的时间间隔。TNS 也是二阶连续的。

图 2.4.8 给出了从相同的点集所生成的 TNS 的三种版本, 但这三个版本在每个节点之间有着不同的时间间隔(由底部的时间线表示)。

为样条路径设定时间表的能力, 使得 TNS 特别适合生成分镜头 camera 路径或飞行, 你可以在任何位置和时间间隔来放置样条的控制点, camera 会精确而又平滑地经过点。

该样条的实现是 SNS 的一个简单扩展。只要为每一段使用指定的时间间隔来代替段长就行。

在图 2.4.8 里最右边的样条, 我们看到, 分配一段较大的时间间隔引起较大的方向偏转。这常常是我们所不希望的, 需要有所改进。为了减少影响, 我们可以在每次平滑后, 对比较

极端的情况，收缩节点速率矢量。下面的收缩效果很好，如图 2.4.9 所示。

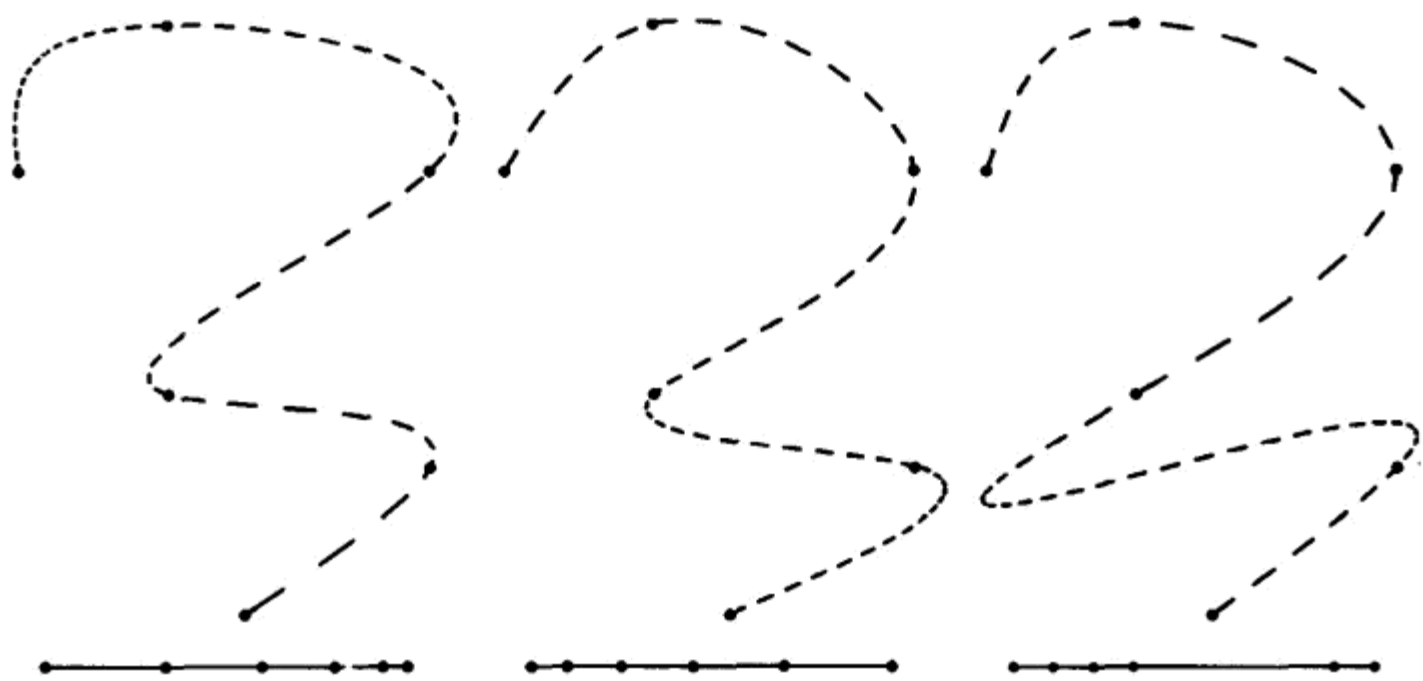


图 2.4.8 在节点上有不同时间间隔的时控非均匀样条

$$v_1^N * = 4r_0r_1 / (r_0 + r_1)^2$$

(2.4.13)

其中 $r = \text{实际段长} / \text{时间间隔}$

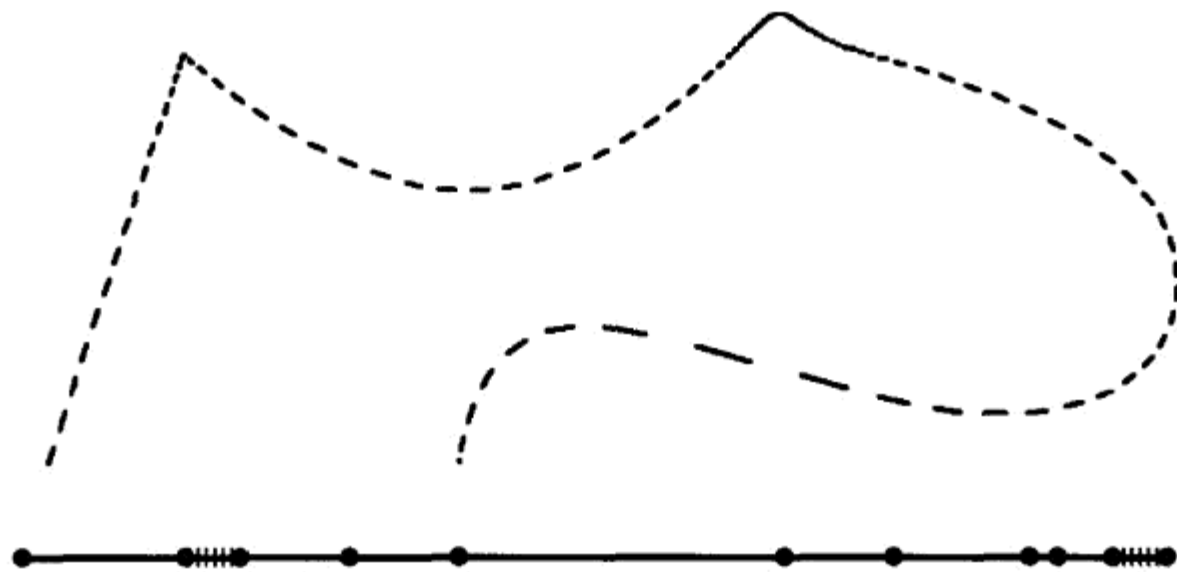


图 2.4.9 受限的 TNS。注意相接近的节点是如何让 camera 平滑地停止和启动的

2.4.6 计算起始和最终节点速率

对于之前描述的样条，这是最后剩下的未定义部分。幸好，有一个相当标准的解法，就是选择速率，以便在终点时的加速度为 0（意味着没有沿曲线方向的力）。看起来，曲线在两个端点的位置开始变直。

在时间 $t = 0$ ，计算公式 2.4.9，我们得到：

$$a^s(0) = 2(-3p_0 + 3p_1 - 2v_0^s - v_1^s) = 0$$

(2.4.14a)

有公式 2.4.5, 所以

$$v_0^N = (3(p_1 - p_0)/l_0 - v_1^N)/2 \quad (2.4.14b)$$

这是我们为样条的起点给出的解。求解终点的速率, 方法基本相似。

$$v_0^N = (3(p_n - p_{n-1})/l_{n-1} - v_{n-1}^N)/2 \quad (2.4.14c)$$

2.4.7 在样条上获取速率和加速度

单考虑某一段的话, 可以用矩阵的形式改写公式 2.4.8 和 2.4.9, 因此, 与公式 2.4.2 联立, 可得到具体的矩阵 H 。

$$H_{velocity} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 6 & -6 & 3 & 3 \\ -6 & 6 & -4 & -2 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad H_{acceleration} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 12 & -12 & 6 & 6 \\ -6 & 6 & -4 & -2 \end{bmatrix} \quad (2.4.15)$$

可以使用标准的查找函数, 但它在段空间返回值。要得到绝对速率, 应用公式 2.4.4; 要得到绝对加速度, 应用公式 2.4.10。

这些额外的量可能非常有用。速率矢量可以为沿着样条路径的对象提供方向。加速度用在诸如对象的倾斜程度, 当前曲率 (curvature) 或推力 (thrust) 方向等物理量上。

2.4.8 优化

对样条访问函数 `GetPosition()` 提速, 有很多办法。该函数对段进行线性的搜索, 尽管是在一个很快的循环中做的, 但时间复杂度还是 $O(n)$ 。如果段长是累积起来的, 那么可以对在所有段上进行二分搜索, 时间复杂度是 $O(\log_2 n)$ 。

在样条被顺序访问而不是随机访问的情况下, 当前段和到起点的距离可以被缓存, 这样查找时间就是常数。此外, 在这情况中, 可以一次性计算和缓存源于公式 2.4.2 的 4×4 矩阵积 HG , 把访问代码减少到和矢量变换一样简单的程度。在具有矢量处理器的硬件平台上, 可以用并行很快地完成。如果样条是用常量 Δt 来计算, 那么我们可以进一步优化每条曲线。可以将函数简化到只作三次矢量加法计算:

```
acc += jerk;
vel += acc;
pos += vel;
```

其中 `jerk` 是曲线的三次导数 (一个常数), 而 `acc`、`vel` 和 `pos` 则是给定的相应初始化值。

2.4.9 结论

本文描述了非均匀样条的三种类型, 它们让游戏开发者在计算样条时能控制样条上的点的时间选择和速率。对于如车辆, 或者像 `camera` 路径等有辅助时控信息的路径, 它们尤其有

用。由于非均匀样条有助于高阶连续或其他的目的，所以对于这些应用而言，它比传统样条要合适得多。

2.4.10 参考文献

[Demidov03] Demidov, Evgeny, “An Interactive Introduction to Splines,” available online at www.people.nnov.ru/fractal/Splines/Intro.htm.

[Hermite99] “Hermite Splines,” available online at www.siggraph.org/education/materials/HyperGraph/modeling/splines/hermite.htm.

[RecipesC93] Press, William H., et al., *Numerical Recipes in C, Second Edition*, Cambridge University Press, 1993, available online at www.library.cornell.edu/nr/bookcpdf.html.



2.5 用协方差矩阵计算更贴切的包围对象

作者: Jim Van Verth, Red Storm Entertainment

E-mail: jimvv@redstorm.com

译者: 许竹钧

审校: 沙鹰

制作游戏时, 我们经常使用包围盒 (bounding box), 或对象的其他简化形式, 来加快对图形和碰撞的相交检查。这些包围对象通常是沿着模型的轴向的, 但这样做不一定能给出严格包围的界限 (也就是说, 最小体积的界面)。有助于解决该问题的一种技术, 是以已知的协方差矩阵 (covariance matrix) 这一方式, 来创建物体的统计测量。加上已知的特征向量 (线性代数的概念), 我们可以用这些来得到一个坐标系, 它对齐于差不多环绕该物体的椭球的长轴和短轴。得到沿着这些轴的最小和最大范围, 会比只用模型坐标系合适得多。

本文的内容覆盖了协方差矩阵及其生成方式, 给出了对特征向量的简单评述, 然后说明了如何为矩阵计算特征向量。最后, 它讨论了如何用生成的坐标轴来生成包围盒和其他包围对象。

2.5.1 协方差矩阵

假设我们将我们的模型视作在 3D 空间由 n 个点 $\{p_1, \dots, p_n\}$ 形成的一片点云 (point cloud), 我们要对点是如何展开、它们在不同的坐标轴如何关联, 做一个统计测量。对于 3D 而言, 这些矩阵已知为在 x 、 y 和 z 中的偏差, 还有在对 xy 、 yz 和 xz 之间的协方差。偏差和协方差通常用来测量相似大小的数据集合的属性——带有年龄、体重和医疗数据的病人列表。在我们的模型中, 我们会将每个点的 x 、 y 和 z 值的集合视作三个等大小, 但可能独立的数据集合来对待。

变量的偏差 (比如 x 坐标) 度量的是数据的点集和平均值的差异, 它计算的是和平均值差的平方和。例如, 我们可以测量 x 坐标的集合 $\{x_1, \dots, x_n\}$ 和平均值 \bar{x} 的偏差。

$$\text{var}_x = \frac{1}{(n-1)} \sum_{i=1}^n (x_i - \bar{x})^2 \quad (2.5.1)$$

在估算中, 我们用 $n-1$ 而不是 n 来纠正偏差。在这情况中, 总的平均值是 n 个点集合的中心, 或者:

$$\bar{p} = \frac{1}{n} \sum_{i=1}^n p_i \quad (2.5.2)$$

所以在 x 方向上:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i \quad (2.5.3)$$

高的偏差意味着数据是展开的, 而低偏差说明数据是集体接近平均值。偏差的平方根是标准偏差。

协方差度量的是数据集合的独立性。例如, 在 x 和 y 坐标的集合之间的协方差可以这样计算:

$$\text{cov}_{xy} = \frac{1}{(n-1)} \sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y}) \quad (2.5.4)$$

x 和 z , y 和 z 之间的协方差可以类似地计算得到。低数量级的协方差意味着数据集合是独立变化的 (如果为 0, 它们完全无关), 但当值高的时候说明它们一起变化。如果协方差为正, 那它们通常一起增加或减少。如果为负, 那么当一个升, 另一个就降。用以前的定义, 在相同数据集合中的协方差等同于偏差: 换句话说, $\text{var}_x = \text{cov}_{xx}$ 。

我们可以整合每个坐标对的偏差和协方差到点集的协方差矩阵 C 。

$$C = \begin{bmatrix} \text{var}_x & \text{cov}_{xy} & \text{cov}_{xz} \\ \text{cov}_{xy} & \text{var}_y & \text{cov}_{yz} \\ \text{cov}_{xz} & \text{cov}_{yz} & \text{var}_z \end{bmatrix} \quad (2.5.5)$$

这是个对称矩阵, 稍后我们会用到这点性质。

为什么这对我们有用呢? 因为, 协方差矩阵在特别的方向给出了对偏差的度量。如果我们用矩阵乘以单位向量, 要是结果向量偏向较高偏差的区域, 它就长些, 偏向较低偏差的区域, 它就短些。拿一个单位圆, 用协方差矩阵来转换它, 会得到一个椭球体。这个椭球的长轴沿着较高偏差的方向展开; 换句话说, 就是在点铺得更开的地方展开。这基本上对齐于我们对象的长轴。类似地, 椭球的短轴沿着较低偏差的方向展开, 也就是在点靠得最紧密的地方, 基本上对齐于我们对象的短轴。因此, 与其用本地轴来创建我们的包围对象, 不如用椭球的轴来代替, 它在很多情况下会更加适合。这些轴就是已知的主轴 (见图 2.5.1)。

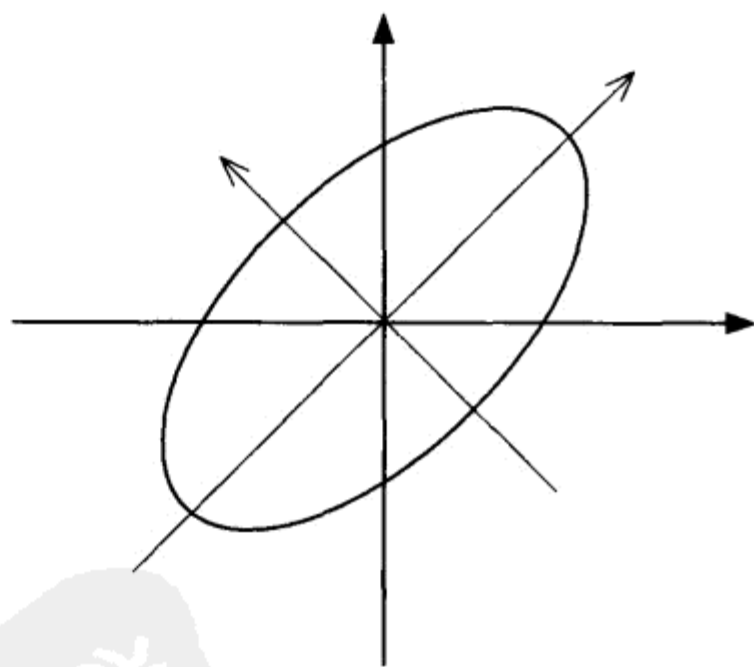


图 2.5.1 显示协方差矩阵的椭球体, 标出了主轴

这样的话, 我们怎么才能找到这些主轴呢? 一个简单例子就是当所有协方差的值为 0 的时候。在这个情况下, 在三个坐标之间数据是完全独立的。我们的主轴对齐于本地坐标轴,

同时一个本地轴指向经过点云的最长距离的方向，一个会指向最短距离的方向。结果得出如下的对角线矩阵：

$$\mathbf{C}_{aligned} = \begin{bmatrix} \text{var}_x & 0 & 0 \\ 0 & \text{var}_y & 0 \\ 0 & 0 & \text{var}_z \end{bmatrix} \quad (2.5.6)$$

可是，通常不用这种方式来组织我们的数据。在这些情况下，我们可以稍微改动矩阵来得到更方便的转换。对于我们正在转换到主轴空间的向量，我们对其进行旋转，并应用协方差矩阵，然后再旋转回来。将此结合到矩阵中，我们就得到了一个对角线矩阵，它类似于前面的矩阵：

$$\mathbf{D} = \mathbf{R}^T \mathbf{C} \mathbf{R} \quad (2.5.7)$$

这个过程称作对角化（Diagonalization）。

由于 \mathbf{R} 是个正交矩阵，列向量组成了坐标系统的正交基。它们等于原始协方差矩阵的主轴，因此我们可以用它们来创建包围对象。下节中，我们来看一下如何用特征向量来得到该旋转。

用以下代码可以从点集中生成平均值和协方差矩阵。由于矩阵是对称的，我们只要计算上三角的值，并把它们存入一维数组中。这没有问题，我们会看到，无需计算下三角的值。我们也不用 $1/(n-1)$ 来规一化矩阵。虽然，这不算一个正确的协方差矩阵，但对于我们的计算而言，那个比例因子无关紧要，而且只会浪费处理时间。

```
void CovarianceMatrix( Vector3* points,
    int numPoints, const Vector3& mean,
    float C[6] )
{
    int i;

    // 计算 mean
    mean = points[0];
    for (i = 1; i < numPoints; ++i)
    {
        mean += points[i];
    }
    float recip = 1.0f/numPoints;
    mean *= recip;

    // 计算矩阵中的每个元素
    memset( C, 0, sizeof(float)*6 );
    for (i = 0; i < numPoints; ++i)
    {
        Point diff = points[i] - mean;
        C[0] += diff.x*diff.x;
        C[1] += diff.x*diff.y;
        C[2] += diff.x*diff.z;
        C[3] += diff.y*diff.y;
        C[4] += diff.y*diff.z;
```

```

        C[5] += diff.z*diff.z;
    }
}

```

2.5.2 特征值和特征向量

为了解决主轴问题，我们需要理解矩阵的特征值和特征向量。假设我们有一个 $n \times n$ 的矩阵 A 。如果有一个非零向量 x ，那么 Ax 就是 x 的标量乘法，或者

$$Ax = \lambda x \quad (2.5.8)$$

接下来我们说 λ 是 A 的特征值， x 是相应的特征向量。换句话说，转换过后，我们向量空间中的某些向量仍然指向相同方向，但它们的长度可能已经改变了。

我们可以将公式 2.5.2 改写成

$$(A - \lambda I)x = 0 \quad (2.5.9)$$

如果，正如我们所说， x 非 0 ，那么两边取行列式得到

$$\det(A - \lambda I) = 0 \quad (2.5.10)$$

这就称为 A 的特征方程。解方程得到所有 λ 的值，我们就得到 A 的特征值。对于 $n \times n$ 的矩阵，特征方程扩展成用 λ 表示的 n 次多项式，我们成为特征多项式。因此，对于我们的例子来说，我们需要解一个三次方程。一般矩阵有复杂的特征值，这是可能的，但由于协方差矩阵是由真正的值组成并且对称，所以它们只有真值表示的特征值。

一个给定的真的特征值，有无数个对应的特征向量。我们把包含这些特征向量的向量空间称为特征空间。解出特定的特征向量，我们用给定的特征值 λ 代入方程 2.5.9，然后解出 x 。

特征值也带几何特性。对有些矩阵，特别是对称矩阵，特征值可以告诉我们矩阵是如何改变向量的值。[Blinn02] 为这种情况给出了一个非常有创见的例子。图 2.5.2 说明了当我们应用矩阵的时候，与单位圆相接的向量是如何改变的。

$$A = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} \quad (2.5.11)$$

椭圆长轴的长度为 4，也就是第一个特征值，并且长轴的方向就是相应特征向量集的方向。短轴一样；长度为 2，或者是我们别的特征值。方向等同于相应特征向量集的方向。

图 2.5.2 应该看起来类似，它和图 2.5.1 相关。如果把图 2.5.1 覆盖到图 2.5.2 上，我们会看到特征向量对齐于主轴。这给了我们一种非正式的提示，可能用计算矩阵的特征向量的方式，能够找到主轴。

注意到对称矩阵还有其他特性，我们可以对此阐述得更加正式些。首先，它们总会有正

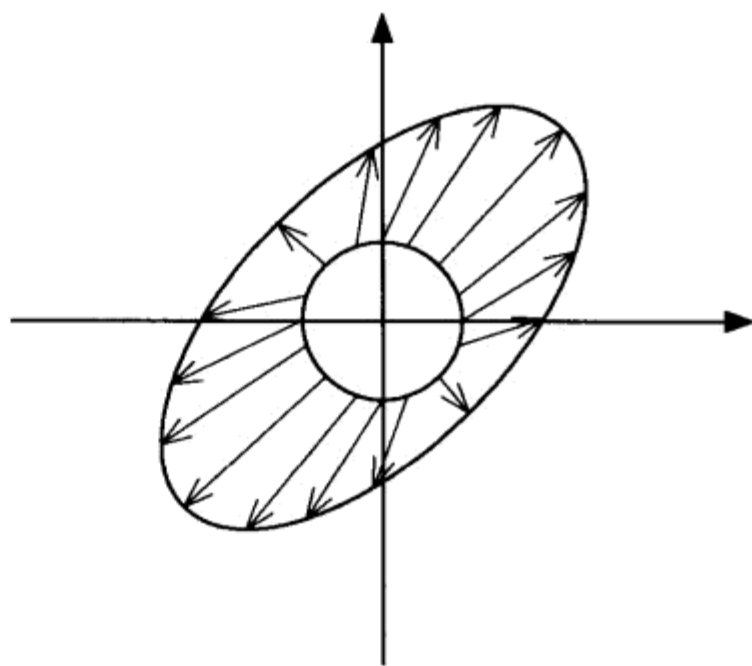


图 2.5.2 矩阵在单位圆上的效果。

沿 (1,1) 和 (-1,1) 方向的向量仍保持原来的方向不变

交向量的集合。其次，它们是正交对角的；更确切地说，对于对称矩阵 A ，我们总能找到一个正交矩阵 R 使得 $R^T A R$ 为对角矩阵。这应该看起来比较熟悉，因为它就是公式 2.5.7。因此，如果我们能找到 R ，我们就能找到我们的主轴。

如何解出 R ？对称矩阵更进一步的属性是， A 的正交特征向量，一旦规一化，就组成了 R 的列。由于在协方差矩阵中，这些列也是主轴，为了找到主轴，我们所需要的就是计算特征向量。因此，我们的直觉是正确的。

2.5.3 计算协方差矩阵的特征向量

对于一般的 $n \times n$ 矩阵，有很多技术可以用来计算特征值和特征向量（参考 [Burden93] 或 [Press93]）。常用的是个普通算法，将矩阵分解成三对角矩阵。然后进一步对角化来计算特征值和特征向量。在 3×3 协方差矩阵中，由于我们不需要这种方法所给予的一般性和精确性，所以这方法就过于繁琐了。实际上，我们可以来计算特征多项式，在这例子中是个三次多项式，然后直接解出方程的根。

矩阵 $C - \lambda I$ 的行列式为

$$\begin{vmatrix} c_{11} - \lambda & c_{12} & c_{13} \\ c_{21} & c_{22} - \lambda & c_{23} \\ c_{31} & c_{32} & c_{33} - \lambda \end{vmatrix} = A\lambda^3 + B\lambda^2 + C\lambda + D \quad (2.5.12)$$

其中

$$\begin{aligned} A &= -1 \\ B &= c_{11} + c_{22} + c_{33} \\ C &= -c_{11}c_{22} + c_{12}^2 - c_{11}c_{33} - c_{13}^2 - c_{22}c_{33} + c_{23}^2 \\ D &= c_{11}c_{22}c_{33} + 2c_{12}c_{13}c_{23} - c_{11}c_{23}^2 - c_{22}c_{13}^2 - c_{33}c_{12}^2 \end{aligned} \quad (2.5.13)$$

我们可以解出这个三项式的根来得到特征值，然后代入每个特征值 λ_i 来计算出矩阵 $M_i = C - \lambda_i I$ 。解出这个线性系统就可以找到相应的特征向量。

$$M_i v_i = 0 \quad (2.5.14)$$

解出三个线性方程式后，我们得到三个特征向量。



[Eberly02] 描述了一种计算这个方程和相应特征向量的高效率的方法。这篇文章的方法已经在子程序 `GetRealSymmetricEigenvectors()` 中实现，在配套光盘中可以找到。它将特征值分成三种不同的情况，根据特例生成特征向量，然后按特征值降序排列。其中数学方面的细节在 Eberly 的文章中有介绍。

2.5.4 创建包围对象

现在我们得到了点云的坐标基（中心和特征向量），就可以计算包围对象了。

要计算包围盒，我们把每个点减去中心，得到差值向量（difference vector），并用每个归一化后的特征向量来做点积计算。相对于每个基向量，这给出了差值向量投影的长度。计算这些点积的最大最小值，就会为点云创建对齐于新的基向量的包围盒。相应代码如下：

```
void ComputeBoundingBox(
    const Vector3* points, int nPoints,
    Vector3& centroid, Vector3 basis[3],
    Vector3& min, Vector3& max )
{
    float C[6];
    CovarianceMatrix( points, nPoints, centroid, C );

    GetRealSymmetricEigenvectors( C,
        basis[0], basis[1], basis[2] );

    min.Set(MAX_FLT, MAX_FLT, MAX_FLT);
    max.Set(MIN_FLT, MIN_FLT, MIN_FLT);

    // 对每个点做如下操作
    for ( int i = 0; i < nPoints; ++i )
    {
        Vector3 diff = points[i]-centroid;
        for (int j = 0; j < 3; ++j)
        {
            float length = diff.Dot(basis[j]);
            if (length > max[j])
                max[j] = length;
            else if (length < min[j])
                min[j] = length;
        }
    }
}
```

注意，中心未必在这个盒的中心，所以最小和最大距离不一定相同。根据意愿，用变换中心的方式可以对此作调整。

圆柱体是包围盒的另一个常见例子。要计算它的话，我们从得到点云的长轴开始，它对应于最大特征值的特征向量。用类似于包围盒的方式可以找到沿轴方向的圆柱体的最大值：我们将每个点和中心的差值投影到圆柱体轴上，然后用轴向量做点积计算。要得到圆柱体的半径，我们来计算每个点和由特征向量和中心组成的轴线之间的距离，半径就是距离的最大值。之前投影的沿轴方向的距离可以重用到这个计算中。

```
void ComputeBoundingCylinder(
    const Vector3* points, int nPoints,
    Vector3& centroid, Vector3& axis,
    float& min, float& max, float& radius )
{
    float C[6];
    CovarianceMatrix( points, nPoints, centroid, C );
```

```

Vector3 v2, v3;
GetRealSymmetricEigenvectors( C, axis, v2, v3 );

min = MAX_FLT;
max = MIN_FLT;
float maxDistSq = 0.0f;

// 对每个点做如下操作
for ( int i = 0; i < nPoints; ++i )
{
    // 沿着轴计算 min, max
    Vector3 diff = points[i]-centroid;
    float length = diff.Dot(axis);
    if (length > max)
        max = length;
    else if (length < min)
        min = length;

    // 计算半径
    Vector3 proj = (diff.Dot(axis))*axis;
    Vector3 distv = diff - proj;

    float distSq = distv.Dot(distv);
    if (distSq > maxDistSq)
        maxDistSq = distSq;
}

radius = sqrtf(maxDistSq);
}

```

可以通过类似的计算得出胶囊状的包围体，只是在两端的半球状盖需要稍多一些计算。更多的细节请参考 [Eberly01] 或 [VanVerth04]。

注意，包围的形状处在由中心和特征向量所定义的形状中。如果我们需要从包围盒空间转换到模型空间，可以用矩阵

$$\mathbf{M}_{BM} = \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \mathbf{v}_3 & \overline{\mathbf{P}} \\ \mathbf{0}^T & & & 1 \end{bmatrix} \quad (2.5.15)$$

2.5.5 结论

本文介绍了一种方法，它用协方差矩阵的特性来计算更合适的包围形状。虽然它比仅用在本地空间中模型的点计算要慢，但它已相当有效率了，而且需要的话，它还可以脱机计算。

但是，它不总是计算包围体积的最恰当方式。有些情况中，对齐模型空间的包围盒仍然是正确的选择。例如，当使用包围盒接近坦克体做快速碰撞，最小体积盒可能相对模型底部沿对角展开。这就让地面测试变得麻烦。

此外, 这技术可能无法计算出最合适的包围形状, 见 [O'Rourke85]。因为协方差矩阵是统计测量, 但数据太过冗余, 凹凸较大、太少, 或很不一致, 就可能引起误差。如果在计算中, 有超过 1 次的三角顶点, 冗余数据就可能发生——基于这个原因, 它最好和三角网格一起计算。至于凹面对象, [Gottschalk96] 建议用物体的凸面来代替数据本身, 可以防止内部的点没有得到测量。

至于稀疏或非一致的数据, 可能的解决途径是, 沿着三角表面得到一致的样本, 并把其作为输入数据。更佳的方式是通过对整个外形作体积分, 生成协方差矩阵, 在无限的分辨率上有效样化模型。这就是惯性张量矩阵 (inertial tensor matrix), 它在计算转换动力的物理模拟中很有用。关于从模型中创建惯性张量的更多信息请参考 [Mirtich96] 或 [Eberly03]。

尽管如此, 在大多数情况中, 在模型的顶点数据上直接使用协方差矩阵, 的确生成非常好的接近, 远好于原始方法。使用该代码或某种推导应该能创建更好的包围盒, 自然而然, 还有更紧密的碰撞和选择检查。

2.5.6 参考文献

[Blinn02] Blinn, Jim, "Consider the Lowly 2x2 Matrix," *Notation, Notation, Notation*, Morgan Kaufmann Publishers, 2002.

[Burden93] Burden, Richard L. and J. Douglas Faires, *Numerical Analysis*, PWS publishing Company, 1993.

[Eberly01] Eberly, David H., *3D Game Engine Design*, Morgan Kaufmann Publishers, 2001.

[Eberly02] Eberly, David H., "Eigensystems for 3×3 Symmetric Matrices," Technical Report, available online at www.magic-software.com, 2002.

[Eberly03] Eberly, David H., *Game Physics*, Morgan Kaufmann Publishers, 2003.

[Gottschalk96] Gottschalk, S., M.C. Lin and D. Manocha, "OBB-Tree: A Hierarchical Structure for Rapid Interference Detection," *Proceedings of SIGGRAPH '96*.

[Mirtich96] Mirtich, Brian, "Fast and Accurate Computation of Polyhedral Mass Properties," *Journal of Graphics Tools*, 1(2): pp. 31–50, 1996.

[O'Rourke85] O'Rourke, J., "Finding Minimal Enclosing Boxes," *Internat. J. Comput. Inform. Sci.*, Vol. 14 (June 1985), pp. 183–199.

[Press93] Press, William H., Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling, *Numerical Recipes in C*, Cambridge University Press, 1993.

[VanVerth04] Van Verth, James M. and Lars M. Bishop, *Essential Mathematics for Games and Interactive Applications*, Morgan Kaufmann Publishers, 2004 (to be published).



2.6 应用于反向运动的雅可比转置方法

作者: Marco Spoerl, KMW

E-mail: mspoerl@gmx.de

译者: 许竹钧

审校: 沙鹰

对于实现游戏中面向目标 (goal-oriented) 的运动, 单纯依靠已经预先创作完成的动画数据通常是不够的。我们总是能在游戏中看见这样那样的问题, 无论是考古学家还是特工等角色, 当他们伸出手, 试图抓住一个对象或按动按钮, 但看上去只是抓住了空气中某个并不实际存在的物体, 或干脆手就伸入了墙里。为了纠正这样的问题, 需要在运行时对动画作调整。这就是实时的反向运动学 (Inverse Kinematics, 简称 IK) 所要做的。

求解反向运动问题的算法有很多种。然而其中的大多数只适用于脱机计算。Chris Welmn 在他的硕士论文中阐述了两类适用于交互环境的方法 [Welman93]。这两种算法中的一种是循环坐标下降 (Cyclic Coordinate Descent, 简称为 CCD), 在另一篇文章 [Weber02] 中已有关于 CCD 的详尽解释。因此本文侧重于 Welman 论文中的第二种算法, 它使用了转置的雅可比矩阵。雅可比转置算法通常能比 CCD 产生更好的结果, 这是因为它总是操作整条反向运动链, 而不是仅观察某一个后继链接。

2.6.1 我们的测试环境

为了简单起见, 本文并不打算用完整的骨骼动画系统来演示雅可比转置。实际上, 我们会使用由相连关节组成的单链。在该形式中每个元素叫做节点。节点的定义为: 一个指向父节点的指针, 指向子节点 (如果存在的话) 的指针数组, 当然还有相对于父节点的转换 (变换和旋转, 但没有比例)。在我们的实现范例中用四元数来表示旋转。

链中的第一个节点称作根, 而最后一个称为末梢 (effector)。末梢应该触摸到目标, 它和所有其他一直到根的节点一起建立了反向运动链, 或称为操纵器 (manipulator)。

图 2.6.1 说明了由 5 个节点组成的反向运动链的初始设置。注意末梢只是用简单的轴来描绘的, 这是因为它没有定义的长度。右边的轴和环绕的球体表示目标和允许到达位置的起始点。

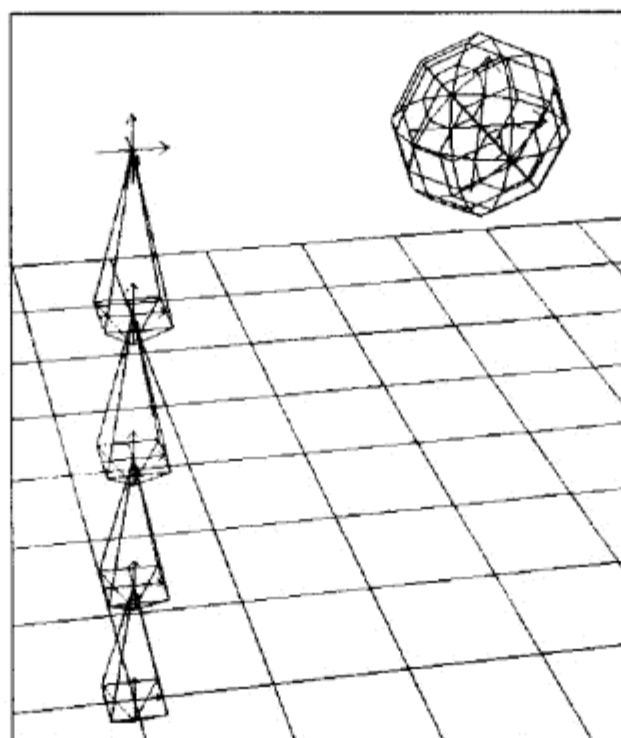


图 2.6.1 操纵器的初始设置



本文中所讨论的应用程序的完整源代码，以及我们用来比较的 CCD 的实现，都可以在配套光盘中找到。

2.6.2 雅可比矩阵是什么？

就正向运动学（Forward Kinematics）而言，末梢的位置 x 可以用以下公式计算得到：

$$x = f(q) \quad (2.6.1)$$

其中 q 表示已知关节变量（例如，旋转和变化）的向量。在比较中，当使用反向运动，目标就是到达已知位置 x 。因此，需要公式 2.6.1 的逆转。

$$q = f^{-1}(x) \quad (2.6.2)$$

巧妙之处在于函数 f 是非线性的。在公式 2.6.1 中，从 q 到 x 存在惟一解，但在公式 2.6.2 中，从一个特殊的 x 到 q 可能有很多映射。要解决这个问题，需要线性化问题。我们用在关节速率和末梢速率之间的关系来达到该目的。

$$\dot{x} = J(q)\dot{q} \quad (2.6.3)$$

关系是由雅可比矩阵，或简称雅可比所给定的，

$$J = \frac{\partial f}{\partial q} \quad (2.6.4)$$

简单地说，雅可比矩阵 J 告诉我们，末梢是如何随着关节变量 q 的变化而改变它的位置和方向。它是个 $m \times n$ 大小的矩阵，其中 n 表示在操纵器中关节的数量，而 m 表示末梢向量 x 的大小。在我们的例子中，就常规意图的位置和方向任务来说， m 为 6。也就是说，雅可比矩阵的第 i 列描述的是，从改变到关节 i ，末梢的位置和方向的增量变化。解决反向运

动问题的一种简单迭代方法，可基于公式 2.6.3 的转换关系得到。

$$\dot{q} = J^{-1}(q)\dot{x} \quad (2.6.5)$$

在实时环境中，我们要避免解方程 2.6.5 所需的昂贵转换。有一种使用伪逆转换的快速方法，但当 J 不是方阵时，这种方法会受到奇异问题的困扰。所以，我们的算法使用了转置的雅可比矩阵。

2.6.3 雅可比转置矩阵简介

反向运动解决的目标就是到达指定的目标。要一直保持“成功”，某种误差测量是必要的。给定当前末梢位置为 $x_e(t)$ ，目标位置为 $x_t(t)$ ，误差测量或距离向量为

$$e(t) = x_e(t) - x_t(t) \quad (2.6.6)$$

现在考虑误差测量 e 是个力向量 f ，它正把末梢拉向理想的目标位置。用那向量 f 我们得到一个在我们操纵器末梢的牵引合力 F 。

$$F = [f_x, f_y, f_z, 0, 0, 0]^T \quad (2.6.7)$$

注意这和 Welman 的版本 [Welman93] 有所不同。

$$F = [f_x, f_y, f_z, m_x, m_y, m_z]^T \quad (2.6.8)$$

由于不处理目标的方向，所以我们忽略了关于某个轴的旋转。然而，根据 [Paul81] 所描述的虚拟工作的原则，在力 F 和内部总的合成力 τ 之间的关系可以描述为

$$\tau = J^T F \quad (2.6.9)$$

所以， τ 代表什么？考虑一个精确的动态仿真， τ 会是关节变量加速度向量。然而，从我们的目的出发，我们将它当作关节速率。

$$\dot{q} = J^T F \quad (2.6.10)$$

最后，对公式 2.6.10 进行积分，我们就得到一个把我们的末梢移向目标的新向量 q 。事实上，积分后， q 带有应用于每个节点的旋转角度。旋转的相应轴是在建立雅可比矩阵时决定的。

计算雅可比矩阵本身也相当简单。由于我们仅使用旋转的关节（每个节点的转换固定），[Welman93] 告诉我们，对第 i 个关节，雅可比矩阵的列项为：

$$J_i = \begin{bmatrix} [(p - j_i) \times \text{axis}_i]^T \\ [\text{axis}_i]^T \end{bmatrix} \quad (2.6.11)$$

其中 p 是末梢的全局位置， j_i 是关节 i 的全局位置， axis_i 是关节 i 的本地旋转轴。简要地说，雅可比矩阵是这样工作的：

- (1) 决定在末梢和目标之间的距离向量和量值；
- (2) 如果距离在一定的门槛之下，忽略计算循环；

- (3) 计算雅可比矩阵和旋转轴;
- (4) 转置雅可比矩阵;
- (5) 决定力 f ;
- (6) 计算关节速率;
- (7) 对关节速率积分, 得到关节旋转并应用旋转;
- (8) 检查退出标准;
- (9) 将最终结果应用于操纵器。

2.6.4 实现算法

我们开始来计算雅可比矩阵本身, 用雅可比转置方法来解决反向运动问题。在我们的实现中, 从公式 2.6.11 得到 p 和 j_i 非常直接。但, 怎样得到轴呢? 由于节点没有固定的轴或旋转, 我们只须使用垂直于两个向量 $\text{joint} \rightarrow \text{target}$ 和 $\text{joint} \rightarrow \text{effector}$ 的向量。

```
for( iColumn = 0, iLinkIndex = 1; iColumn < iLevel; ++iColumn, ++iLinkIndex )
{
    // 当前节点的位置
    m_arrCurrentTM[ iLinkIndex ].GetTranslation( vecLink );

    // 向量当前连接 -> 目标
    vecLinkTarget = m_vecTargetPosition - vecLink;

    // 向量当前连接-> 当前末梢位置
    vecLinkEnd = vecEnd - vecLink;

    // 计算轴
    m_arrAxis[ iColumn ] = CVector3::CrossProduct( vecLinkTarget, vecLinkEnd );
    m_arrAxis[ iColumn ].Normalize();

    // 计算雅可比矩阵项的上三角部分
    vecEntry = CVector3::CrossProduct( vecLinkEnd, m_arrAxis[ iColumn ] );

    // 设置雅可比矩阵的项
    for( iRow = 0; iRow < 3; ++iRow )
    {
        m_arrJacobian[ iRow * iLevel + iColumn ] = vecEntry[ iRow ];
        m_arrJacobian[ ( iRow + 3 ) * iLevel + iColumn ] = m_arrAxis[iColumn][ iRow ];
    }
}
```

注意, $iLevel$ 表示操纵器中连接的数目; 也就是说, 雅可比矩阵中项的个数。下一步, 当然, 需要转置雅可比矩阵。

```
for( iRow = 0; iRow < 6; ++iRow )
    for( iColumn = 0; iColumn < iLevel; ++iColumn )
        m_arrJacobianTransposed[ iColumn * 6 + iRow ] =
            m_arrJacobian[iRow * iLevel + iColumn ];
```

来自公式 2.6.7 的力 f 就是在末梢和目标之间的距离。注意，这个向量还有第二个用途，就是可以用它的量值来推出计算循环的标准。

```
// 决定末梢的位置
m_arrCurrentTM[0].GetTranslation( vecEnd );

// 要达到目的必须做的是什么?
vecDifference = vecEnd - m_vecTargetPosition;

// 有多远?
fError = vecDifference.GetMagnitude();

// 设置力向量
farrForce[0] = vecDifference.GetX();
farrForce[1] = vecDifference.GetY();
farrForce[2] = vecDifference.GetZ();
farrForce[3] = 0.0f;
farrForce[4] = 0.0f;
farrForce[5] = 0.0f;
```

利用公式 2.6.10，很容易计算得出关节速率。

```
// 计算 q'

for( iRow = 0; iRow < iLevel; ++iRow )
{
    m_arrQDerivate[ iRow ] = 0.0f;

    for( iColumn = 0; iColumn < 6; ++iColumn )
        m_arrQDerivate[ iRow ] +=
            m_arrJacobianTransposed[ iRow * 6 + iColumn ] * farrForce[ iColumn ];
}
```

完成基本的计算后，我们必须对关节速率做积分，并应用所产生的改变。

```
// 积分并应用改变

for( iIndex = 0, iLinkIndex = 1; iIndex < iLevel; ++iIndex, ++iLinkIndex )
{
    axisAngle.SetAxis( m_arrAxis[iIndex] );
    axisAngle.SetAngle( m_arrQDerivate[iIndex] * 0.01f );
    quatAlign.SetQuaternion( axisAngle );
    quatAlign.Normalize();


    // 存储的元数
    ...
}
```

你可以看到，我们使用了简单的欧拉积分步长。尽管它不是积分的最佳方式，但它一定是最快的。我们让读者来试一下较为复杂的方法，比如在 [Welman93] 中提出的龙格—库塔

(Runge-Kutta) 积分器。

最终，结果值是用作指定关节的旋转角，而关节在早先计算好的轴周围。

之前提起过，我们将四元数用于旋转。在雅可比转置的情况中，保持轴一角的一致使用会更佳。在这种情况下，来自积分的结果的应用会变得简单，而且无需之前所说明的转换。在我们的实现范例中，尽管雅可比转置解决器和一般的 CCD 解决器共享一个通用类，但那个 CCD 解决器过多地使用了四元数。

 完全的雅可比转置实现在配套光盘可以找到，位置在文件 CIKSolver
ON THE CD JacobianTranspose.cpp 中的 CIKSolverJacobianTranspose::Solve() 函数中。

2.6.5 结果和比较

由于我们使用了欧拉积分步长来解方程 2.6.10，所以当选择步长大小的时候，需要小心谨慎。用我们的实现范例在几个不同值之间作的比较见图 2.6.2（迭代的最大值为 100）。

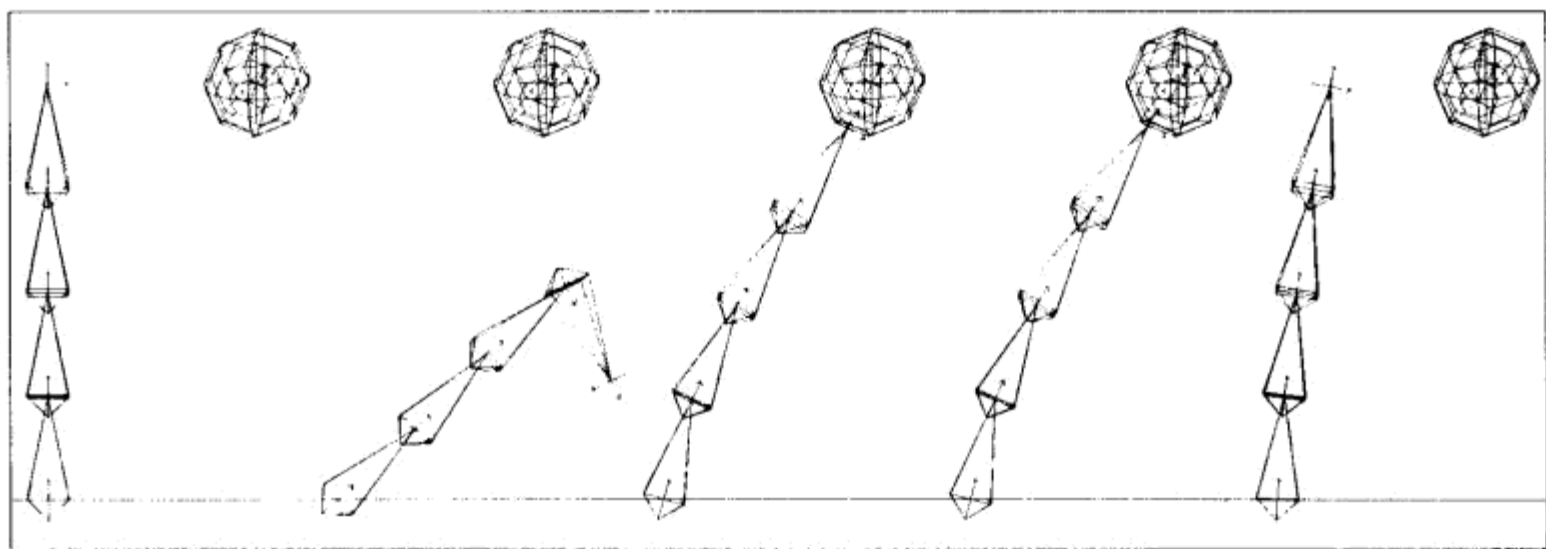


图 2.6.2 欧拉积分步长比较。

从左到右分别对应于：初始设置、0.1、0.01、0.001 和 0.0001

步长为 0.1，无论对迭代的限制做怎样的改动，算法一点都不会收敛。当使用 0.0001 的步长，为了接触目标，不得不增加迭代的次数。注意，目前 0.01 的步长被用于以下所有测试。

首先，让我们比较一下由雅可比转置 (JT) 和从 CCD 的一般实现所得结果的不同。

在第一种情况中，目标就位于末梢的右下方。链由 7 个对旋转没有限制的节点组成。迭代限制设为 100。图 2.6.3 说明了在两种算法之间的不同。尽管雅可比转置在操纵器上均匀分布旋转，CCD 法单独考虑每个连接。

来自图 2.6.4 的设置几乎等同于之前的一个，除了在这个情况中，沿着主轴的旋转被限制在 ± 50 度内。我们看到 CCD 的结果改善了，但 JT 结果看起来不变。注意到，在我们的实现范例中，只用了一个简单的欧拉角度限制方案。

图 2.6.5 说明了如果有个更严格的限定 (± 20 度)，两种方法都无接触目标，CCD 看起来又被改进了。

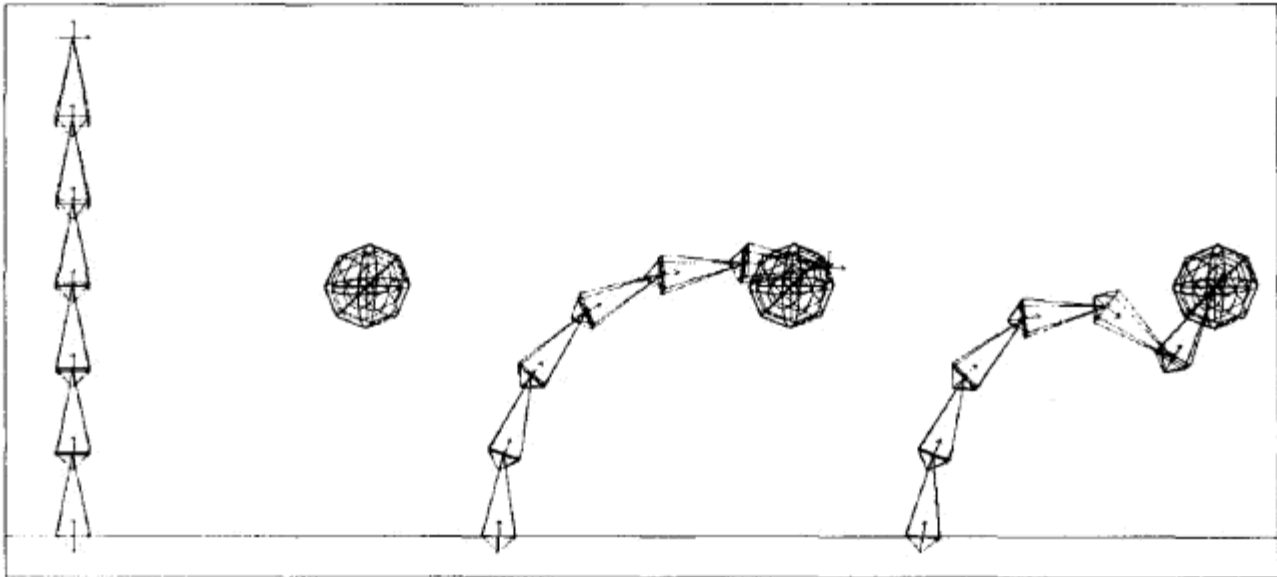


图 2.6.3 一个简单非约束的设置。从左到右：初始状态、JT 和 CCD

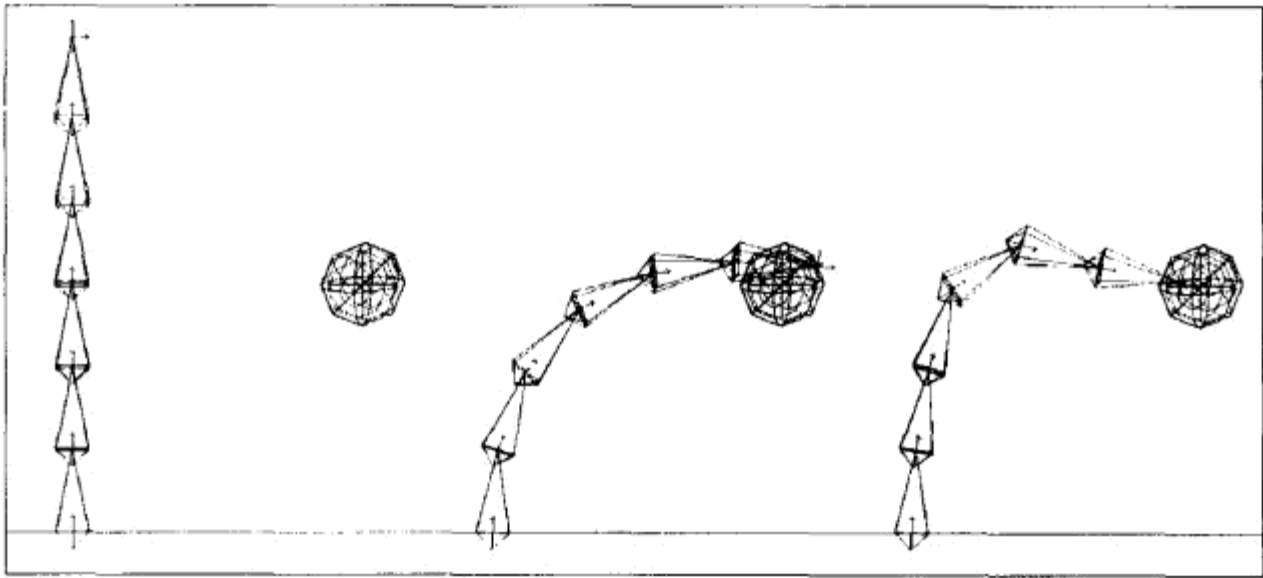


图 2.6.4 一个简单受约束的设置。从左到右：初始状态、JT 和 CCD

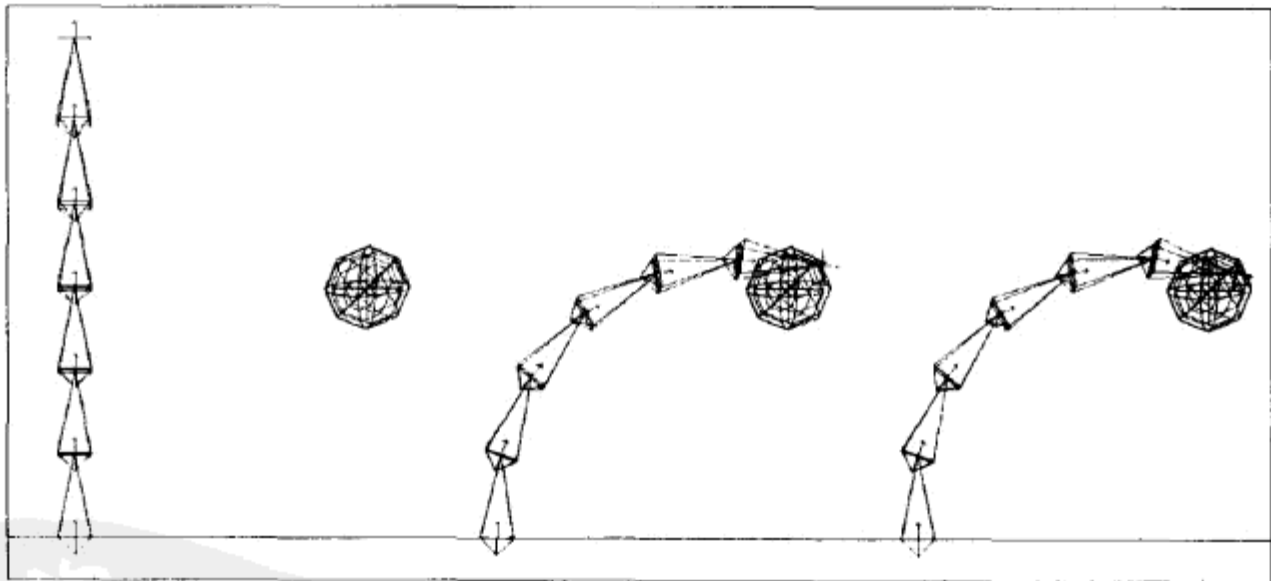


图 2.6.5 另一个受约束的设置，有更严格的限定。从左到右：初始状态、JT 和 CCD

我们的第二个测试案例使用了更复杂的设置（见图 2.6.6）。目标位于远离末梢的地板上。链有 9 个旋转不受约束的 9 个节点。再一次，迭代的限定设为 100。当 JT 计划平滑的接触目标的时候，CCD 给出了相当混乱的解决方案。

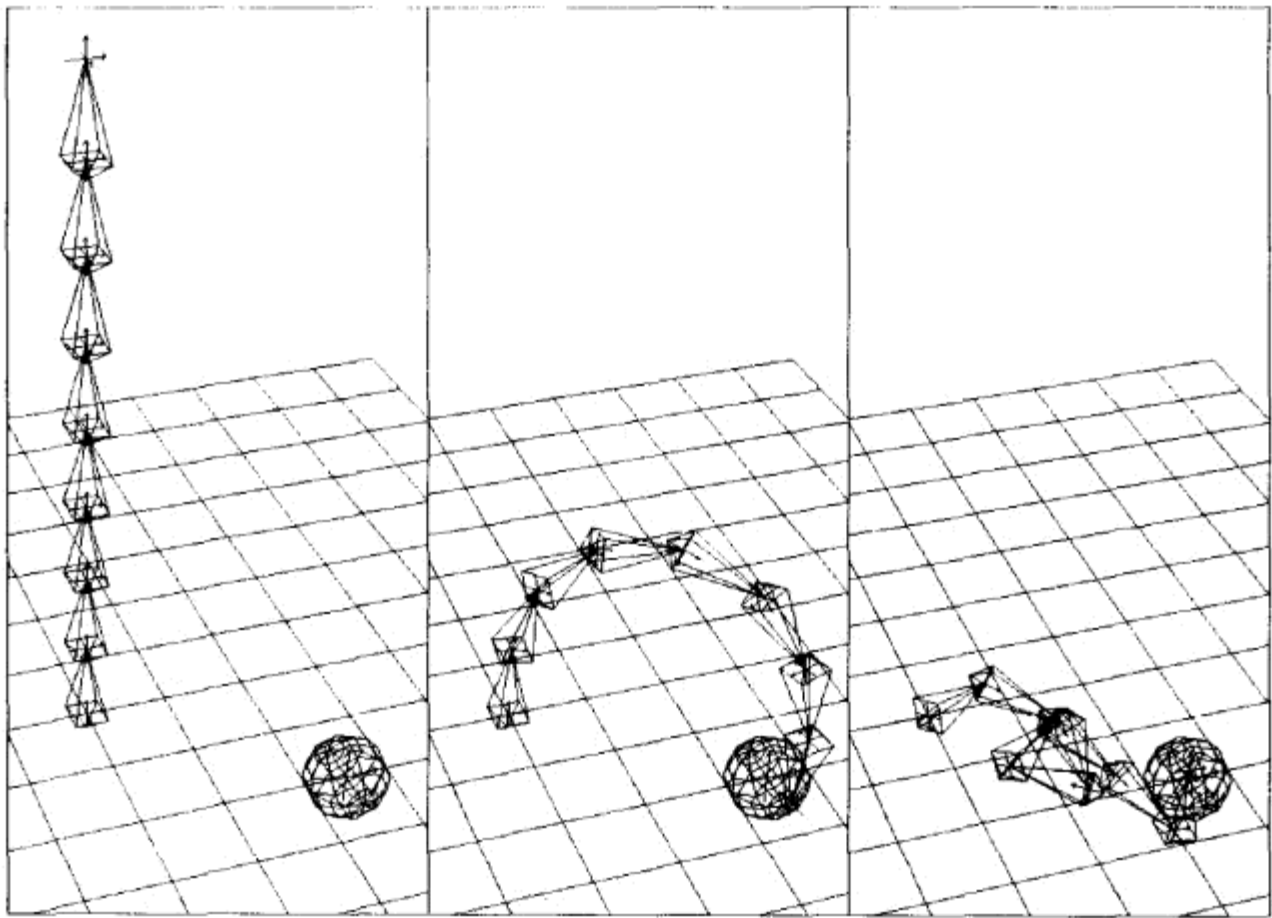


图 2.6.6 一个没有约束的复杂设置。从左到右：初始状态、JT 和 CCD

在二维的测试中，旋转的角度在 ± 50 度内。此外，图 2.6.7 说明在这个情况中，虽然 CCD 更精确，但不尽人意；而 JT 仍然应付自如。

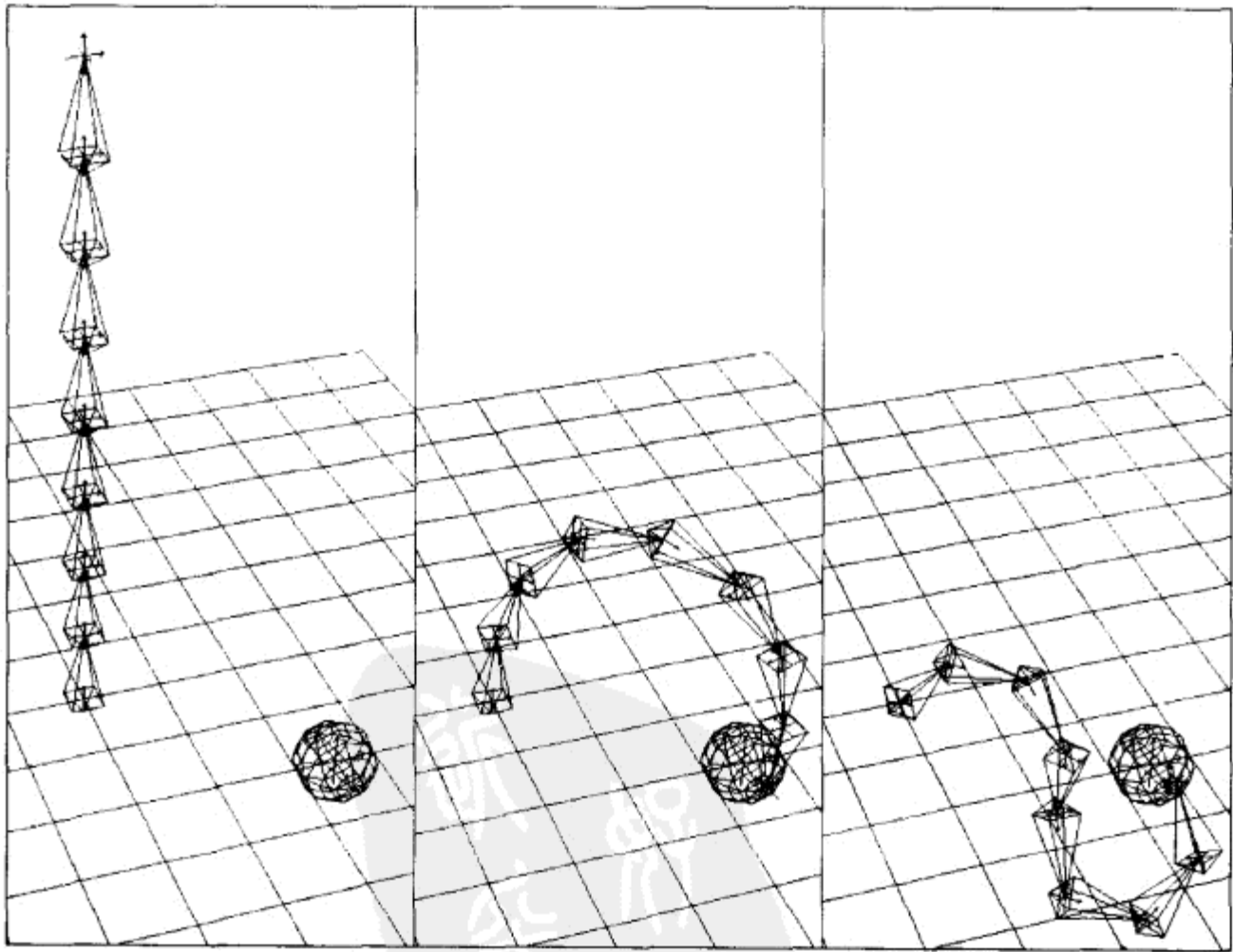


图 2.6.7 一个有约束的复杂设置。从左到右：初始状态、JT 和 CCD

最后，图 2.6.8 所示的比较证明了在使用严格限定 (± 20 度) 的情况下，CCD 的效果也改善了。

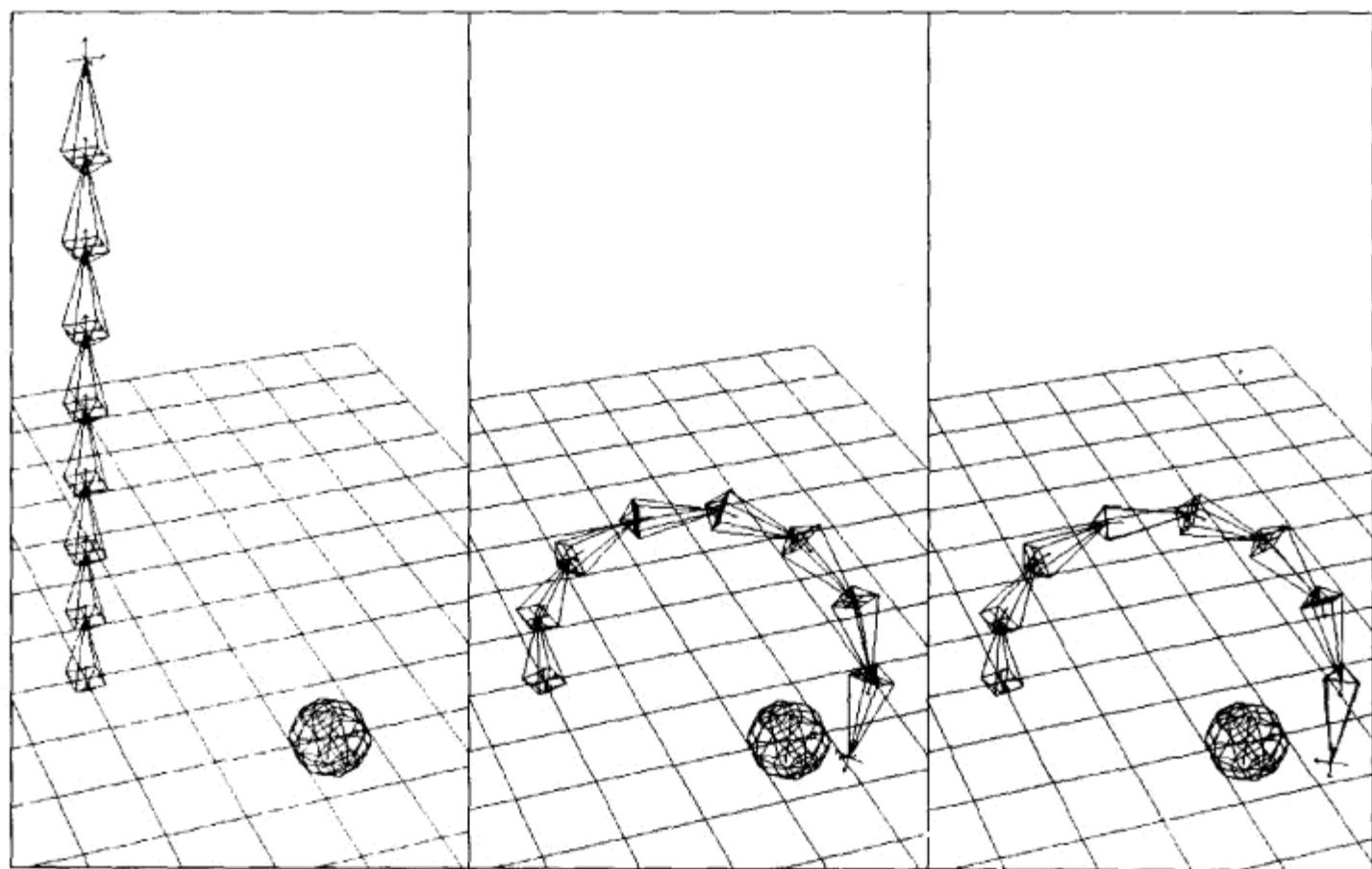


图 2.6.8 另一个带有更严格限定的复杂设置。从左到右：初始状态、JT 和 CCD

2.6.6 结论

我们为循环坐标下降 (CCD) 提供了一种强有力的备选技术。前者在非约束环境和使用较大旋转限制的时候是主要采用的方法。后者均匀分布的节点旋转则提供了很好的效果。

然而我们仍未涉及到约束自由度的方式。在我们的实现范例中，为了简明起见，在记录结果四元数之前，只有四元数到欧拉转换是用来限制旋转（用 pitch/yaw/roll 限定）的。注意，由于雅可比为每一步都重新计算，所以这在收敛标准上没有任何明显的影响。我们强烈鼓励读者阅读 [Weber02] 或 [Blow02] 中讨论限制旋转的较高级的方法，以及 [Welman93] 中有关于合并约束的一章。

最后，还要提到有关目标方向匹配的问题。正如之前所提到的，为了有利于简化的公式 2.6.7，我们在 Welman 的原始公式 2.6.7 中忽略了切力。要扩展本文所表示的方法，可以确定一个扭矩以将操纵器的方向设定在目标的方向上，并在作用于操纵器末梢的力中加入相应的轴。

2.6.7 参考文献

[Blow02] Blow, Jonathan, "Inverse Kinematics with Joint Limits," *Game Developer* Vol. 9, No. 4 (April 2002): pp. 16–18.

[Paul81] Paul, R.P., *Robot Manipulators: Mathematics, Programming, and Control*, MIT

Press, 1981.

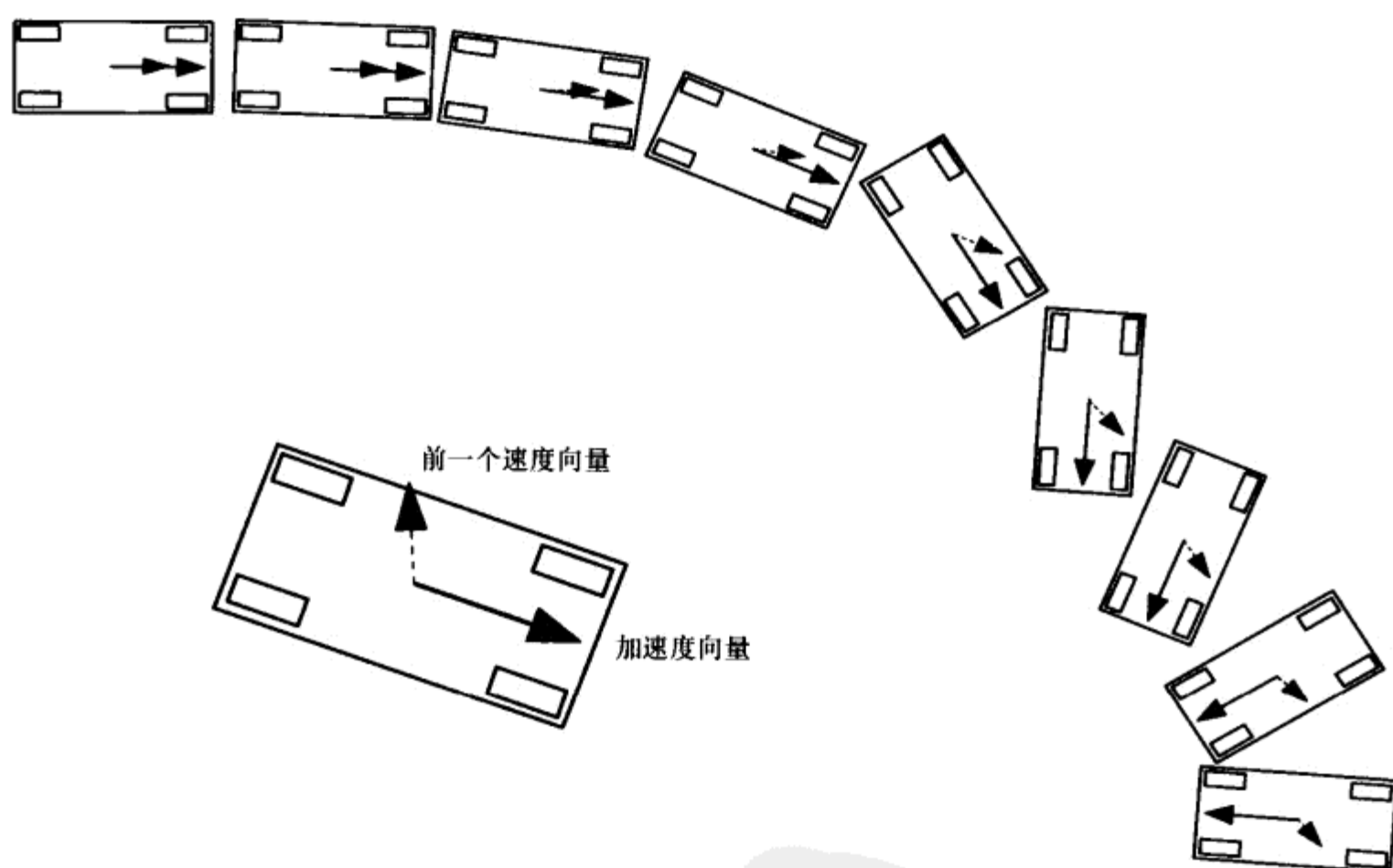
[Schreiber98] Schreiber, Guenther, and G. Hirzinger, "Singularity Consistent Inverse Kinematics by Enhancing the Jacobian Transpose," German Aerospace Center—DLR, available online at www.robotic.dlr.de/Guenter.Schreiber/ark1998.pdf.

[Weber02] Weber, Jason, "Constrained Inverse Kinematics," *Game Programming Gems 3*, Charles River Media, 2002.

[Welman93] Welman, Chris, "Inverse Kinematics and Geometric Constraints for Articulated Figure Manipulation," Master's Thesis, Simon Fraser University, September 1993.



物 理



简介

作者: Graham Rhodes

E-mail: grhodes@nc.rr.com

译者: 李鸣渤

审校: 沙鹰

多年以来,我一直对已故的著名物理学家理查德·费曼心存景仰。而我仰慕他的原因,并不是由于他在物理领域做出的卓越贡献,而是因为他除了对物理问题,对生活的细节也同样用心。他爱玩,不会错过每一个探索新体验的机会。他对游戏十分热爱,有一次在加利福尼亚理工学院(Caltech)的一个派对上,他用自己如猎犬一般敏锐的洞察力,在不知道大家的选择的情况下,准确地将派对参与者与他们各自所选取的书一一对应,取悦了在场的所有人们;他在 Los Alamos 国家实验室从事原子弹研究期间,热衷于研究撬锁和破解保险柜的学问(为了指出它们设计上的漏洞,让锁着的东西更安全);他还独自运用解谜方法,破译了现存的三部玛雅象形文字古抄本之一《德累斯顿抄本》。费曼学过绘画,还曾是一位优秀的爵士乐曼波鼓演奏者。

在我接触过的所有领域中,游戏开发是最需要人具有费曼那样的对学习、体验和游戏的渴望的。近几年来,物理已经成为游戏开发中重要的领域之一。实时物理(real-time physics)一向都是飞行模拟游戏的核心,在某种程度上也是赛车游戏的核心。然而现在,物理的重要性正在向整个游戏领域发展延伸。在我看来,在游戏中运用实时物理主要有三个好处。第一(基本上也是最重要的),在有合适的流水线工具(pipeline tool)的情况下,使用物理可以减少必须由美工部门创作的动画的数目,从而显著减少动画制作的成本。(特别是在人物角色动画方面,虽然通过动作捕捉得到的(motion-captured)和由美工制作的关键帧动画(keyframe animation)在给角色塑造个性上仍然起着举足轻重的作用,然而,它们一旦与基于物理模拟的解决方案结合使用,则能创造出无限的动作行为。)第二,物理可以模拟角色、对象与游戏世界之间的任意互动,从而使得仅仅采用美工制作动画无法有效实现的“突现行为”(emergent behavior)成为可能。

运用实时物理的第三个好处是因为它很酷。今天,游戏迷们经常在网上论坛中讨论游戏里布偶角色物理(ragdoll character physics)和可互动的水的模拟有多么好。很快,讨论的话题就会变成有关柔体动力学(soft body dynamics)和完全由物理驱动的角色。不过,我相信终有一天,使用实时

物理的游戏不再会让人有惊羡的感觉。但是至少现在，使用物理可以突出你的优点，而且记住继续使用物理能削减开发成本并且制作出更棒的游戏。

欢迎阅读首次和读者见面的“游戏编程精粹”系列物理部分！从这里开始，您将会发现有益于游戏开发的各种主题。首先，Roger Smith 和 Don Stoner 根据军事实验的结果提出了一套估计命中率的公式。合理地运用这些公式能大大改进游戏的攻击判定！接着 Marcin Pancewicz 和 Paul Bragiel 介绍了一个很不错的简化车辆物理模型，它小而快，适合手持游戏机。他们的方法也可以用于俯视视角的陆上、海上以及空中交通工具的模拟。接下来连续三篇文章，主题是具有复杂的碰撞反应（collision response）的刚体动力系统的开发。首先，Nick Porcino 展示了一个基于 Verlet 积分的面向对象刚体物理引擎，这个引擎可以作为基于物理模拟游戏的基础。然后 Russ Smith 直观地讨论了有关刚体约束（rigid-body constraint）的数学问题，吹散了笼罩在复杂机械模拟和身体互动表现上的层层迷雾。Adam Moravanszky 和 Pierre Terdiman 则展示了几个如何在碰撞检测中简化多组碰撞点的方法，从而增加了碰撞反应计算的稳定性（并可能减少计算开销）。换个口味，Jerry Tessendorf 阐述了一个称为 iWave 的水波模拟方法，它能很好地支持水波与物体之间的相互作用。最后，还有两篇文章介绍了模拟柔性物体的新技术。Thomas Di Giacomo 和 Nadia Magnenat-Thalmann 阐述了一个用于柔性物体模拟的多层的方法，它可用于模拟各类柔体变形，快速而且稳定。最后，James O'Brien 讲解了基本的模式分析，它可用于游戏中柔性物体的实时模拟，无论模拟的时间间隔（simulation time step）还是帧速率（frame rate）如何，都具有百分之百的稳定性。

虽然用于物理的开发工具在不断的进步，但是使用起来仍然困难，而且物理引擎还不够强大和可靠。我希望读者能利用这些文章，提高自己的游戏引擎，并开发全新的东西。说不定，在这个过程中，您能创造游戏的未来！



3.1 死神的十指：战斗中的命中算法

作者：Roger Smith 和 Don Stoner, Titan Corporation

E-mail: roger@fingersofdeath.com, don@fingersofdeath.com

译者：李鸣渤

审校：沙鹰

好的射击类游戏需要好的命中算法。这篇文章介绍了一系列的“死神之指（finger of death）”——命中算法，它可以用来增强攻击决策的真实性。文章中所讨论的大部分算法都是为美国军方所开发的，并且至少经过一次实战模拟的验证。

虽然现有的第一人称视角射击类游戏的命中算法也是不错的，但是，在某些情况下，把几何、统计、概率和集合运用进去，能使结果的计算更准确、更高效。在大型多人游戏和即时战略游戏中有着许多行动，无需逐个用视线算法（Line Of Sight, LOS）建模，也不需要达到爆头（headshot）的精确度。这些基于实弹射击实验所得出的多种命中类型的算法，可为其他游戏所利用。在第一人称视角射击类游戏中，开发者可以把这些算法用在电脑控制的敌人身上，而对于游戏者控制的游戏角色（avatar，化身）使用较精确的 LOS 算法。

3.1.1 射击带状物（Ribbon）

死神之指第一指：决定一名射击者能否命中一个带状目标，如公路、河流、车流，或长条形生物（见图 3.1.1）的简单方法。如果目标太长，没有办法射击到其顶端或尾端，命中目标的概率就只能取决于目标的宽度以及武器射击模式的标准偏差了。这个简单的算法也是在除了以下要介绍的几种损耗算法 [Parry95] 以外，运用逻辑和数学的好方法。

射击命中点的偏差取决于许多因素，如武器的质量、使用者的稳定性和技术、各种不同弹道发射器的机械结构以及风速风向情况。

命中带状物体的概率（ P_h ）公式如下：

$$P_h = 2W / \sqrt{2\pi\sigma_x} \quad (3.1.1)$$

其中：

W 是目标的宽度；

σ_x 是 x 方向分散的标准偏差。

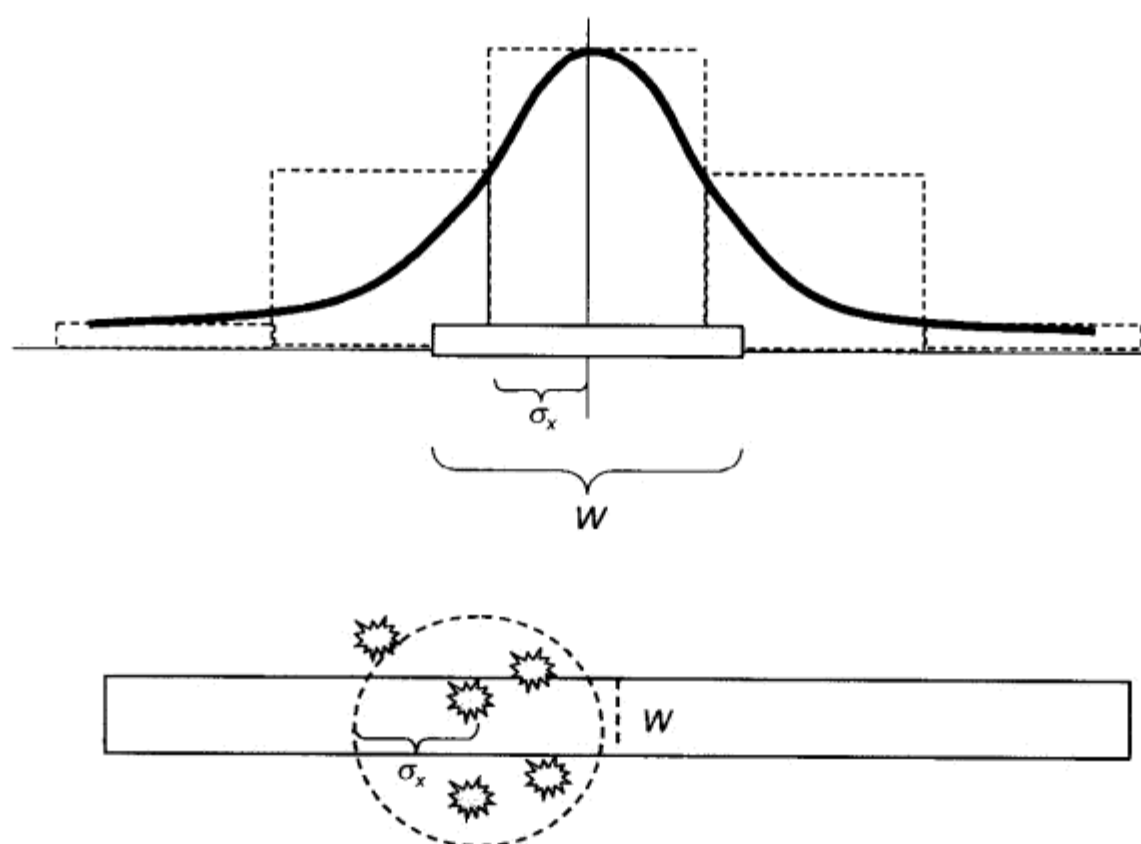


图 3.1.1 命中带状目标的概率

这里假设这种模式分布在 x 方向和 y 方向上的标准偏差公式都是一致的。



以上所述所有算法的代码都可以在随书所附的光盘中找到。

3.1.2 射击靶心

死神之指第二指：介绍了命中一个圆形目标的概率。象前面讲到的算法，基于的事实是：所有射击者，人以及机械，在每发出一次射击时都不同。

这个算法由两个十分简单的变量组成——目标的半径以及圆形的标准偏差。这个偏差基于一个平均值为 0 的正态分布（normal distribution），因为射手是直接对目标的中心进行瞄准 [Parry95]。虽然算法能决定每一击能否命中目标，但是却计算不出实际的弹着点。这一简化消除了对于圆形 x 和 y 轴分布的计算。

计算命中圆形物体的概率（ P_h ）方程式如下：

$$P_h = 1 - e^{-(r^2 / 2\sigma_x^2)} \quad (3.1.2)$$

其中：

r 是目标的半径；

σ_x 是 x 方向子弹分散的标准偏差。

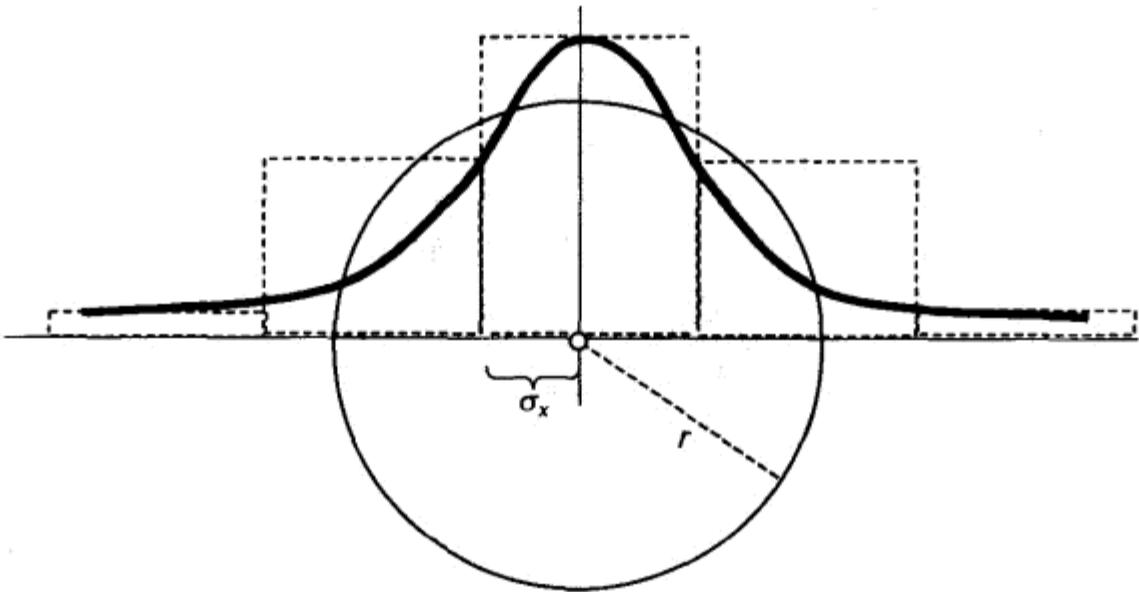


图 3.1.2 命中靶心的概率

3.1.3 射击矩形

大部分目标的形状都不是圆形的，因此，我们需要一个更灵活的算法，用于射击方形目标，比如人的躯干或汽车。这个算法包含了一个方形目标的长度和宽度尺寸[Parry95](见图 3.1.3)。

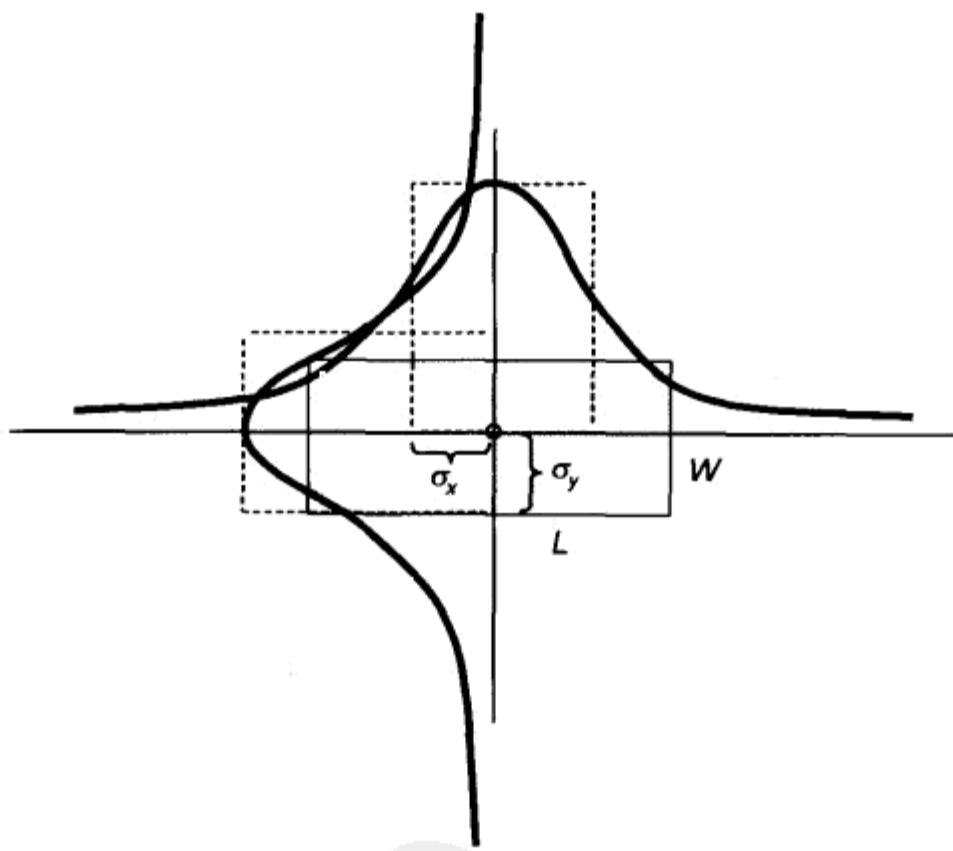
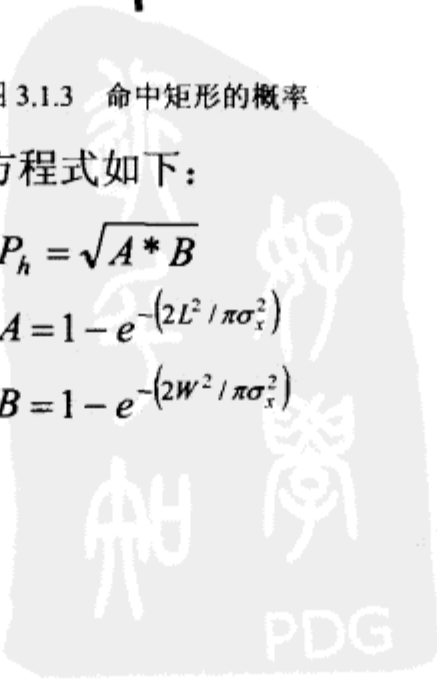


图 3.1.3 命中矩形的概率

计算命中方形物体的概率 (P_h) 方程式如下：

$$\begin{aligned} P_h &= \sqrt{A * B} \\ A &= 1 - e^{-(2L^2 / \pi\sigma_x^2)} \\ B &= 1 - e^{-(2W^2 / \pi\sigma_y^2)} \end{aligned} \tag{3.1.3}$$

其中：



L 是目标在 x 方向的长度;

W 是目标在 y 方向的宽度;

σ_x 是 x 方向子弹偏差的标准偏差;

σ_y 是 y 方向子弹分散的标准偏差。

武器在 x 和 y 方向的标准偏差常常是不一样的。比如, 当一名足球四分卫传球的时候, 投掷点沿球飞行方向坐标轴的变差一般就比投掷点两边的偏差大一些。向战斗机车发射导弹或者向恐龙投掷石头时的情况也是这样的。

3.1.4 使用霰弹枪射击小目标

有些武器能一次性发射无数火箭、小炸弹或者爆炸性武器, 以泰山压顶之势完全镇住目标, 并把它打成碎片。在这种情况下, 就要用更快的方法计算整个霰弹群的整体杀伤力, 而不是分别计算每一颗霰弹的杀伤力, 然后把它们相加。

这个算法计算的是霰弹的有效杀伤范围覆盖点目标的概率, 而目标的大小在计算中则不予考虑, 因为我们假设有效杀伤力范围覆盖了整个目标 [Parry95] (见图 3.1.4)。

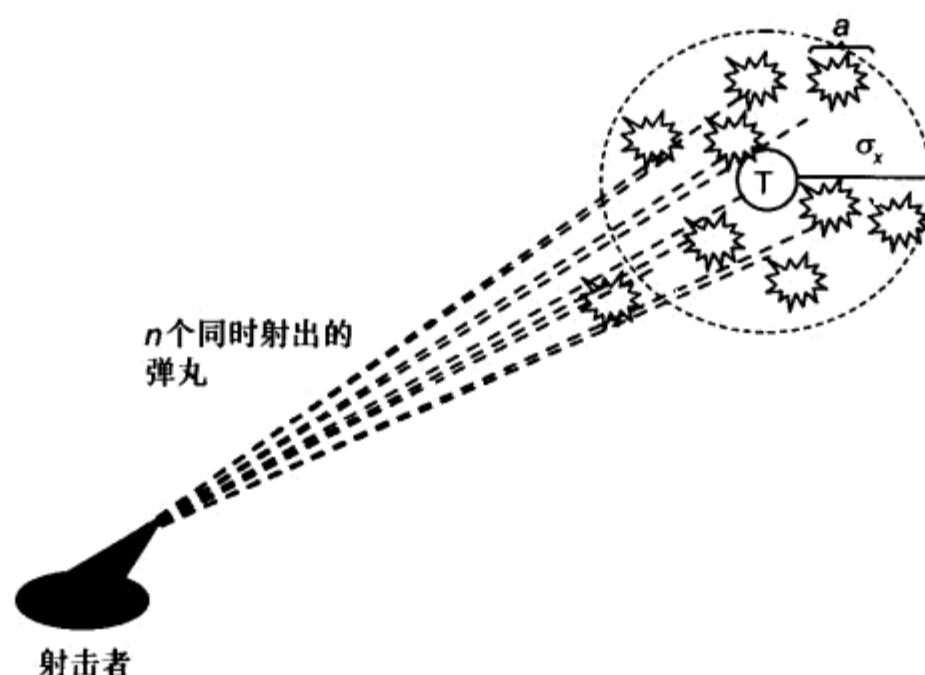


图 3.1.4 同时发出弹丸射杀目标的概率

$$P_k = 1 - e^{-na/2\pi\sigma_x^2}$$

其中:

n 是弹粒的数量;

a 是每一颗弹粒对目标的有效杀伤范围;

σ_x 是 x 方向子弹分散的标准偏差。

3.1.5 移动炮兵的攻击命中

为了对目标进行下一轮更准确的攻击, 炮兵经常会根据前方侦查部队所发回的敌目标所

在位置的信号来修正射击方向。这样的攻击要比前一种霰弹的杀伤力要强。这种攻击方法的计算是通过计算系列的指数求和得出的 [Parry95] (见图 3.1.5)。

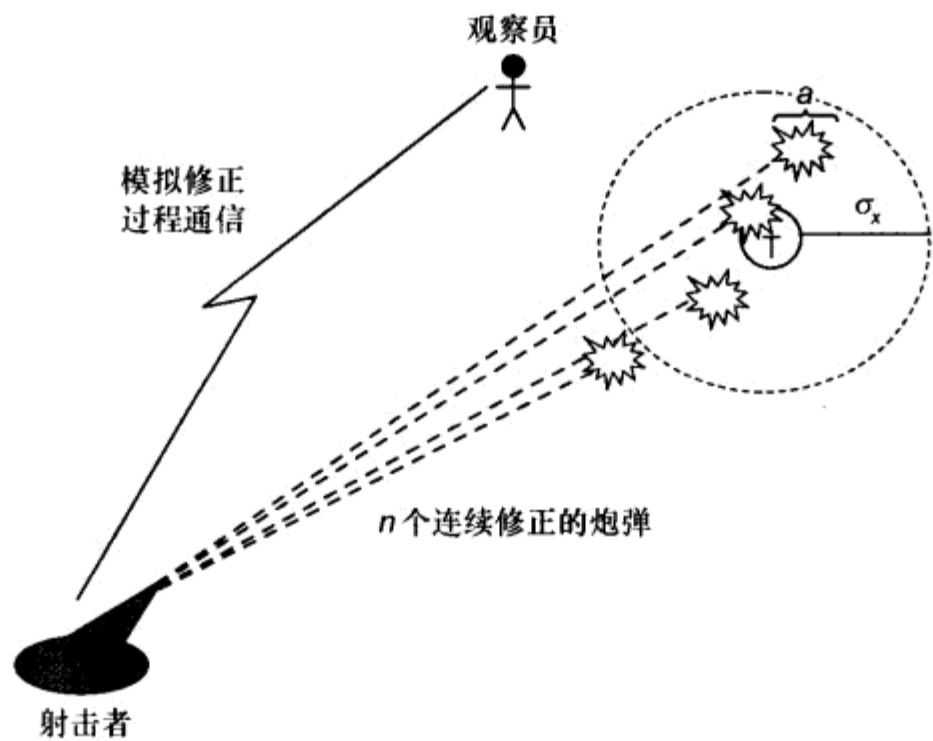


图 3.1.5 有目标修正观察员移动火炮的致命性

$$P_k = 1 - e^{-\left(\frac{a^2}{2\sigma_x^2}\right) \sum_{i=2}^n \frac{(i-1)}{i}} \tag{3.1.4}$$

- 其中：
- n 是弹粒的数量；
 - a 是每一颗弹粒对目标的有效杀伤范围；
 - σ_x 是 x 方向子弹分散的标准偏差。

3.1.6 死亡的 4 种主要形式

公猪 Napoleon 曾经说过：“所有死亡都是平等的，只是有些和其他的相比更公平。”^①军事模拟通常对真实世界的战斗中最经常看见的 4 种不同类型的死亡形式建模。第一种主要形式是机动性死亡 (mobility kill)，即目标不能发生移动了，但还有能力发射武器或与其他车辆通信。第二种是火力死亡 (firepower kill)，即武器毁坏了，但是车辆或人还能够动。第三种是移动加火力死亡，即机车没有完全损坏，人尚存活，但是不能移动或使用武器。这样的目标可能还能观察敌人的行动、通信、军需补给，而且在一些模拟中，还能启动援救行动。最后一种死亡方法就是灾难死亡 (catastrophic kill) 或简称 K-kill，形象地说：一架飞机爆炸成上百万的碎片、着火的坦克炮塔在空中旋转，或士兵粉身碎骨，都是属于 K-kill。

这 4 种死亡形式可以用维恩图表示 (Venn diagram) (见图 3.1.6)。虽然这一形式能清楚

^① 译者注：原话是英国作家乔治·欧威尔在他的《动物农庄》一书中通过 Napoleon the pig 之口说出的 “All animals are equal, but some animals are more equal than others.”

地表达死亡类型之间的关系，但是为了便于应用，应该把它分离，这样可以很快地决定每种情况用哪种死亡类型。这样分离的数据通常表示为死亡指示计（见图 3.1.7）。在指示计表现的单个空间内使这些死亡形式标准化使得一个程序可以通过随机抽取一个号码来决定交战的死亡类型。

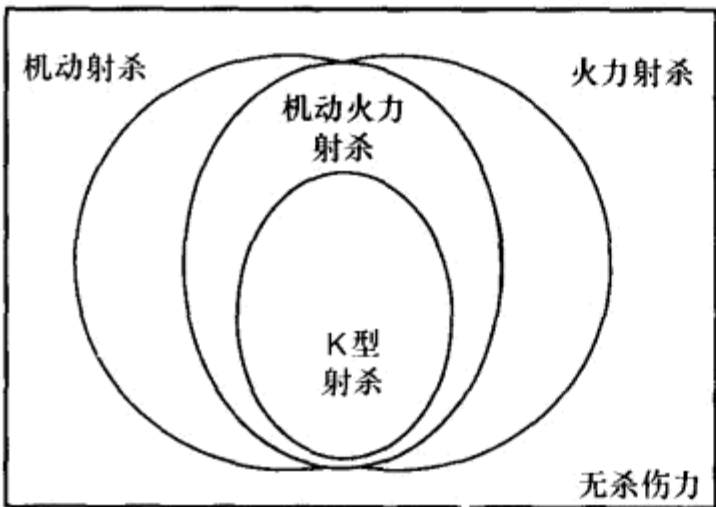


图 3.1.6 标准射杀类型

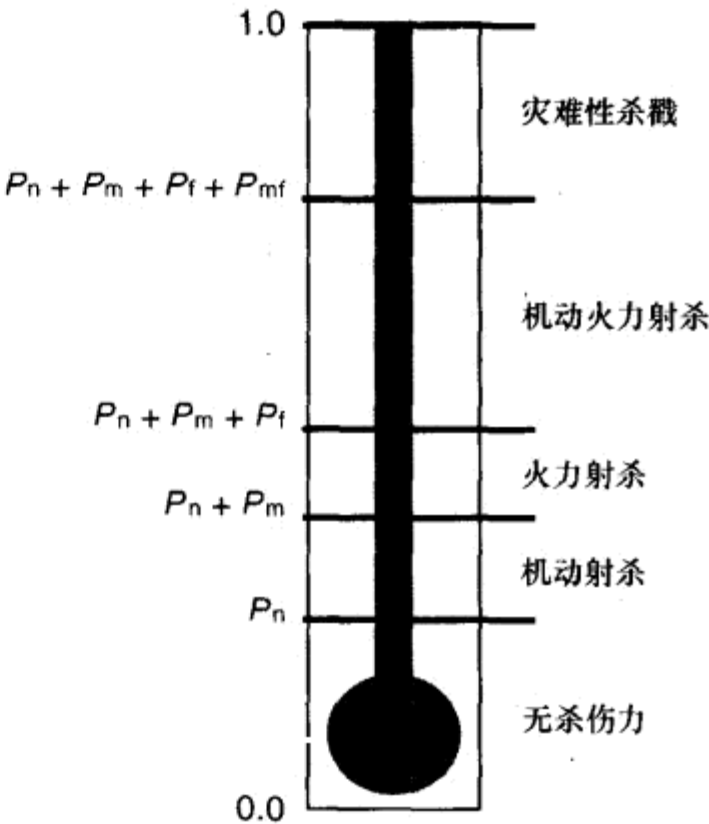


图 3.1.7 射杀温度计

实弹射击通过向真实的目标发射真实武器、测量结果来决定不同情况下每种死亡类型的可能性。大多数模拟和游戏不能得到这么丰富的信息资源。因此，我们必须从试验数据中识别出一般趋势，建立模拟的方程式，同时保持足够的灵活性，能应用于新的武器/目标中。模拟项目能发现从实弹试验中得到的机动、火力及灾难死亡数据间独特的关系。这一关系使其可以建立简单的方程式，使用单个“机动或火力死亡概率”的基数（图 3.1.7 中所有阴影部分）来计算其他所有概率。他们发现在受伤后机动死亡发生的概率是 90% ($P_m = 0.9 * P_{MoF}$)；火力死亡发生的概率为 90% ($P_f = 0.9 * P_{MoF}$)；而灾难死亡的概率为 50% ($P_k = 0.5 * P_{MoF}$)。

可是，这些资料不能直接应用在图 3.1.7 中的死亡标示计上。 P_m 并不是说 90% 的交战结果会导致机动死亡，它是说 90% 的机动或火力死亡包括了机动死亡。那么，我们要把它分开，使其可以抽取一个任意数字，决定对目标应用哪种死亡形式。这些独立的死亡概率为以下所示。

$$\begin{aligned} P_n &= 1.0 - P_{MoF} \\ P_m &= P_{MoF} - P_F = 0.1 * P_{MoF} \\ P_f &= P_{MoF} - P_M = 0.1 * P_{MoF} \\ P_k &= 0.5 * P_{MoF} \\ P_{mf} &= P_{MoF} - P_m - P_f - P_k = 0.3 * P_{MoF} \end{aligned}$$

(3.1.5)

其中，下标指出发生哪一种类型的死亡概率。例如， P_m 是仅遭遇机动死亡，而没有任何其他死亡形式的概率。 P_n 是不发生死亡的概率。



这些独立的死亡概率决定了最终落在一个死亡指示计中点在哪里。随书附送的光盘中的代码有一个应用示例。

3.1.7 化学武器、火球及区域性魔法

计算化学武器及其他同类武器偏差的模型已经有不少了。下面要介绍的这个简单的算法可以通过释放的化学物品的数量及释放发生地离目标的距离来计算死亡的概率。在游戏中，这一算法可以用于膨胀的火球、区域性魔法或任何其他怪异的大杀伤力武器。

$$P_k = \left(\sqrt[3]{nw_r} / \sqrt{2\pi} \right) * e^{-0.5 * (k * d^2 / nw_r)^2} \tag{3.1.6}$$

其中：

- n 是攻击到某一点的弹粒数；
- w_r 是每弹粒内化学品的重量（以公斤为单位）；
- d 是弹粒从目标位置掉下的距离（米为单位）；
- k 是一个常数，说明了化学物品的偏差特性，在这个试验中，我们觉得从 0.00135 值开始为佳。

这个方程式使我们可以对每一个弹粒分别处理，或者计算若干弹粒以同一个撞击点为中心的单个攻击。方程式还加入了代表化学混和品密度及粘度的常数 k 。你可以调整这个值来达到想要的效果。

3.1.8 弹片的楔入

导弹一般很少会直接命中飞机。导弹一般会到达一个“最近接近点”，然后在飞机附近引爆。之后，导弹的碎片从爆炸点呈环状或球状散开，飞机被弹片命中继而被摧毁 [Ball85]。这个算法可以用于爆炸性投射武器、火球，及魔法来瞄准飞机、龙及其他敌人（见图 3.1.8）。

$$x = nA_v / (2\pi r^2 (\cos \phi_1 - \cos \phi_2)) \tag{3.1.7}$$

$$P_k = 1 - e^{-x}$$

其中：

- n 是导弹头的碎片或射弹数；
- A_v 是目标相对于导弹的攻击面，以平方米为单位；
- r 是从爆炸点到目标的距离，以米为单位；
- ϕ_1 是从导弹弹道到最近目标攻击范围边缘的角度；

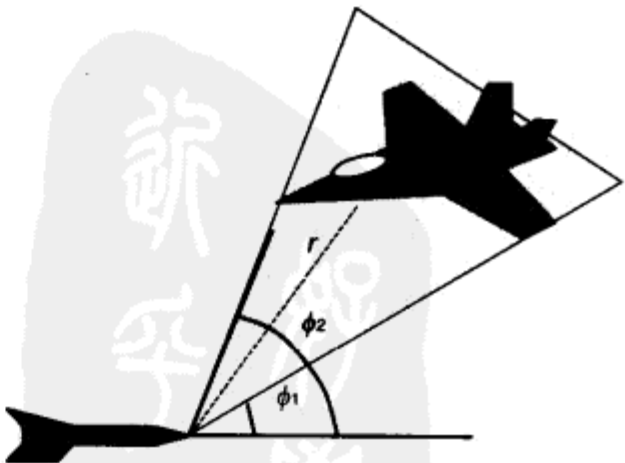


图 3.1.8 弹片楔入的射杀概率

φ_2 是从导弹弹道到最远目标攻击范围边缘的角度。

3.1.9 攻击丛林

有些交战中有成队的猎人在领地或丛林中搜索隐藏起来的猎物 [Shubik83]。当一大队猎人寻找一大队猎物的时候，可以把猎物的追捕猎杀看作一个整体来模拟，而不用去表现每个猎物的单独行动、每个猎人和猎物的视线范围。跟前面一样，当追捕猎杀是由人工智能猎人操作，特别是在屏幕外发生的时候，这种方法很有用。

这种算法是为计算根据猎人数量及效率决定改变猎物的数量而设计的。它也能解决不同类型的猎物和捕猎者，例如，小型啮齿动物、中等大小的狼，及大型的象。

要运用这种算法，我们必须定义在已知条件下（开放区域、森林、城市等）每种类型的猎人相对每种类型的猎物侦测的可能性。我们还需要一个参数来决定猎物躲避、逃跑，或逃过猎人能力的“生存力”因素。这些数据通常是通过试验和观察决定的（见图 3.1.9）。

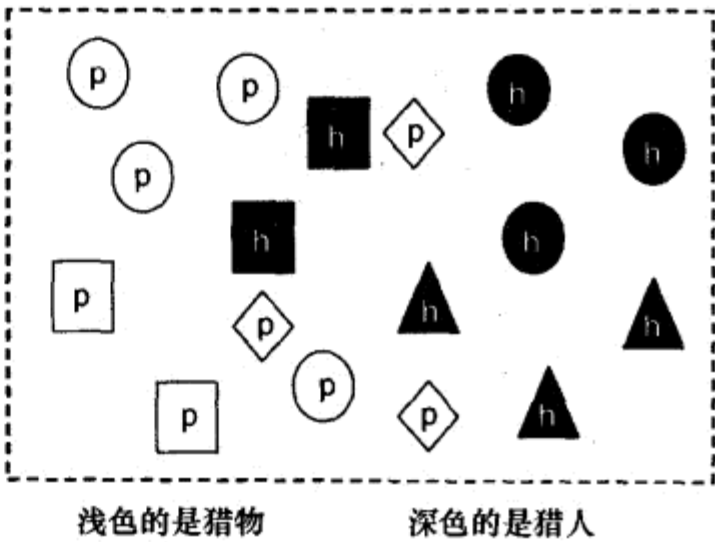


图 3.1.9 多类型猎人寻找多类型猎物

$$x = \left((k_j / p) * \sum_{i=1}^n D_{i,j} * h_i \right)$$
$$A_j = p_j * (1 - e^{-x})$$

(3.1.8)

- 其中：
- A_j 是 j 类动物被猎杀的数量；
 - p_j 是 j 类猎物的数量；
 - k_j 是在 $[0, 1]$ 范围内猎物的生存能力参数；
 - p 是各种猎物的总数量；
 - n 是猎物的种类数；
 - $D_{i,j}$ 是 j 类猎物被 i 类猎人发现的概率；
 - h_i 是 i 类猎人的数量。

3.1.10 攻击有猎物分布的丛林

最后一条死亡规则是对前一条的修正。数学家和分析家发现前一种算法不能解决隐藏在丛林中猎物的不同密度。很明显，搜索区域内有一百头猎物的时候找到并猎杀猎物比搜索区域内只有两三头时要容易得多。因此，有关人士想出了一个称为 Lulejian 模型的变体 [Shubik83]，其中猎物间的间隔是一个很重要的因素。这种算法得出的可视画面和前面介绍的是一样的，但是在数学上根据解决猎物间的距离而有所不同， k_j 的定义也有细微差别，Lilejian 把 k_j 定义为猎人对 j 类猎物的平均攻击力。

$$x = \left(\sum_{i=1}^n k_j * h_i \right) / (s * p) \quad (3.1.9)$$

$$A_j = p_j * (1 - e^{-x})$$

其中:

A_j 是 j 类动物被猎杀的数量;

p_j 是 j 类动物的数量;

s 是搜索区域内猎物间的平均距离, 以米为单位;

p 是各种猎物的总数量;

n 是猎物的种类数;

k_j 是在 $[0, 1]$ 范围内猎人对 j 类猎物的平均攻击力;

h_i 是 i 类猎人的数量。

3.1.11 结论

这篇文章中介绍的死神之十指（十条命中规则）只是应用在计算机游戏中的战斗命中算法的冰山一角。在其中使用的有关几何、概率、统计和物理的概念都展示了解决问题的较好方法。游戏开发人员应该像军事模拟人员一样来做改进——运用经验、数学、创造力及其他科学，找出适用于自己游戏的方程式。千万不要害怕试验！

3.1.12 参考文献

[Ball85] Ball, Robert E., *The Fundamentals of Aircraft Combat Survivability Analysis and Design*, AIAA Press, 1985.

[Parry95] Parry, Samuel, editor, *Military OR Analyst's Handbook: Conventional Weapons Effects*, Military Operations Research Society, 1995.

[Shubik83] Shubik, Martin, editor, *Mathematics of Conflict*, Elsevier Science Publishers, 1983.



3.2 在低速 CPU 系统中交通工具的物理模拟

作者: Marcin Pancewicz, Infinite Dreams; Paul Bragiel, Paragon Five

E-mail: highway@idreams.com.pl, paul@paragon5.com

译者: 李鸣渤

审校: 沙鹰

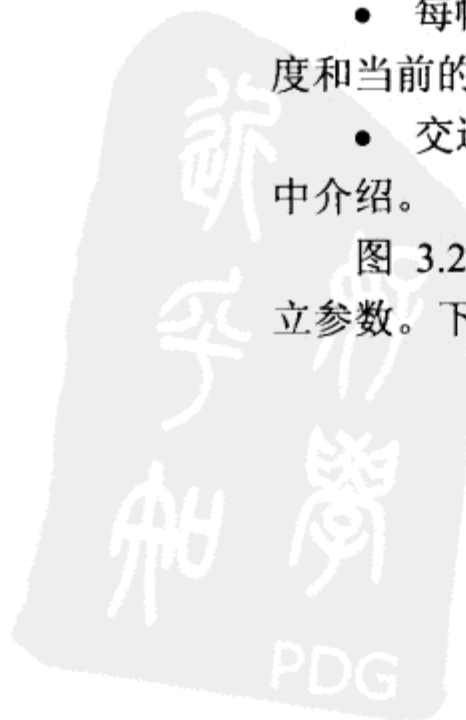
在开发我们的第一款俯视视角赛车游戏的过程中, 我们遇到了一个问题, 那就是如何为 CPU 性能有限的系统开发快速灵活的物理引擎。我们要求这个引擎能够实现各种各样的交通工具, 从雪地摩托和赛车, 到直升飞机和气垫船等非陆地交通工具。在这篇文章中, 我们将具体介绍这样的一个专为系统资源非常有限的手持式游戏机而设计的引擎。希望读者在理解文中介绍的算法后, 能在文中解决方案的基础之上, 制作出自己的物理引擎。

3.2.1 技术的概要和前提假设

仔细观察运动中的交通工具, 你将会发现, 在交通工具行驶中有两个最基本的要素: 通过油门和刹车系统控制的加速度, 通过方向盘控制的行驶方向。通过分别模拟这两个要素, 我们可以用一套计算量很小并且简化过的物理规则来建造一个灵活性很强的系统。为了简化计算, 我们需要作下列假设。

- 踩油门只在当前的行驶方向上产生加速度; 而踩刹车则会产生方向与当前行驶方向相反的加速度。
- 每帧的模拟时长 (Time Step) 设为 1.0, 因此简单地将前一帧的速度和当前的加速度相加, 就可以得出当前帧的速度。
- 交通工具可以自由地旋转。这个我们会在后面的“方向控制”一节中介绍。

图 3.2.1 中显示了用来模拟交通工具的移动、转向和加速度的各个独立参数。下面我们讨论如何模拟加速度以及方向控制。



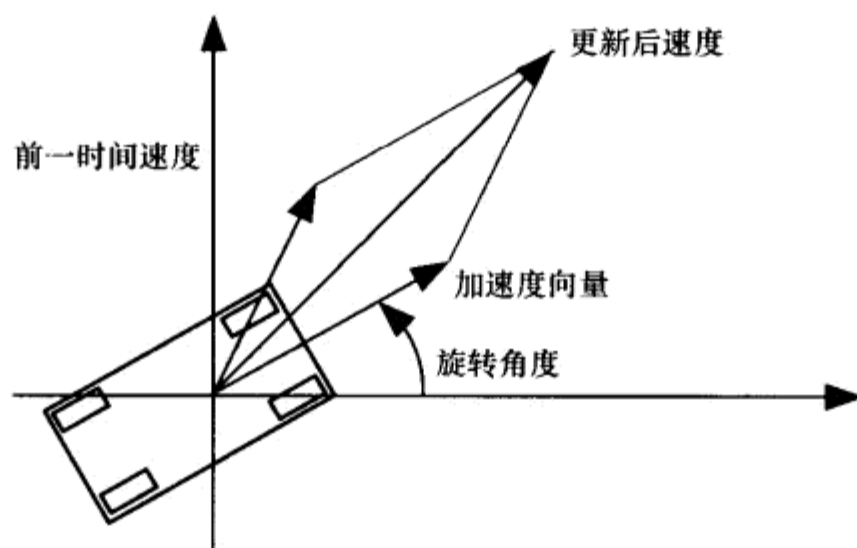


图 3.2.1 有关交通工具运动的简化参数

3.2.2 交通工具沿当前行驶方向上的加速及减速

虽然决定交通工具加速度的直接因素是作用在交通工具上的力，但是为了简化物理计算，我们可以根据需要假设交通工具在加速时受到某种特定条件的限制。

我们已经知道，在初速度为零的匀加速运动中，当前的速度等于加速度乘以时间：

$$v = at \quad (3.2.1)$$

其中， v 表示速度， a 表示加速度， t 表示从初始时刻起交通工具的运行时间。

但是，匀加速在游戏中并不常用。为了可以在游戏中应用非均匀加速，我们需要求如下方程的积分，并每帧将积分结果与当前速度 v 相加。

$$v'(t) = a \quad (3.2.2)$$

（撇号表示对时间的导数。例如，速度对时间的导数就是加速度。）下面这个方程式使用了显性欧拉方法（explicit Euler technique）对方程 3.2.2 进行数值积分。在前面一节中讲过，在所有相关方程式的计算中，都将每帧的模拟时长假设为 1.0。这不仅是为了简化计算，还能减少所涉及到的浮点计算次数。如果每帧的模拟时长不是 1.0，加速度 a 在和前一帧速度相加之前必须乘以每帧的模拟时长。

$$v_n = v_{n-1} + a \quad (3.2.3)$$

这个方程式不单适用于所有理想的驾驶环境，而且也适用于交通工具受外力作用情况下的非理想驾驶环境。在实际运用时，只要在计算加速度 a 的时候把外力考虑在内就行了。显性欧拉积分法虽然不是最精确的，但是用在这样的简单引擎中已经足够准确了，并且速度非常快。另外，在忽略弹力（springlike force）的情况下，它也该算是相当稳定的了。

为了实现更真实的运动模拟，我们在计算时还要把摩擦考虑进去。在本文讨论的引擎中，我们假设在每次迭代计算时，都按某个常系数使物体的速度匀速降低。虽然严格说来这不是正确的物理定义，但是，就我们的目的来说已经足够安全了。根据需要，我们可以用 f 来代表摩擦，虽然它并非代表某一个摩擦力。 f 的值根据车辆当前所在的地形变化而变化。

这样，这个方程式就变成了：

$$v_n = f v_{n-1} + a \quad (3.2.4)$$

在这个方程式中，速度 (v) 和加速度 (a) 是矢量，摩擦系数 (f) 是标量。

计算 a 时，在不考虑外力的情况下，方程式 3.2.4 中的加速度表示牵引力或者发动机的扭矩，或者刹车所产生的加速度；这几个因素我们将在下一节进行讨论。构造加速度矢量模型最简便的方法就是使用极坐标。用极坐标，可以很容易地把你的交通工具引向你所期望的方向（用一个角度 rot 来表示），并可把对应加速度的力表示为等于矢量长度的单个值。从极坐标系换算到普通的笛卡尔坐标系并不难，请参见图 3.2.2。

我们可以看出，这段代码非常简单。

```
// 首先我们把用笛卡尔坐标系 x 的 2D 向量 Vnml
// 乘以表示摩擦力大小的 F

Vn_x = Vnml_x*F; // nml means n-1
Vn_y = Vnml_y*F;

// 然后加上使用极坐标表示的加速度向量

Vn_x += A*cos(rot);
Vn_y += A*sin(rot);
```

现在，踩下加速踏板（油门），车子就开始加速；松开油门，车子就减慢直至停下。

模拟输入设备

有一个小问题：在大多数的手持游戏机上，加速踏板（油门）是由单个按钮控制的，这个按钮只有按下和松开两个状态。用实际驾驶打个比方，就像是给你一辆巨大结实的 Trans Am 跑车，可是不巧油门不是全开就是全闭。要是真的开上这样一辆车，每次踩下加速踏板，驾驶员的身体就会猛地后仰，非把脖子甩坏了不可。

要解决这个问题，方法之一就是 把或开或关的加速器模型改为随着加速按钮按下时间的增长而逐渐加速，按钮松开后逐渐减速的加速器模型。同时，还必须设置初始值和上限，这样才能把加速度控制在一个合理的范围之内。可以通过反复调节，以达到更为真实的效果，比如使提速快、减速慢，或者提速慢、减速快。图 3.2.3 反映了相关的信息。图

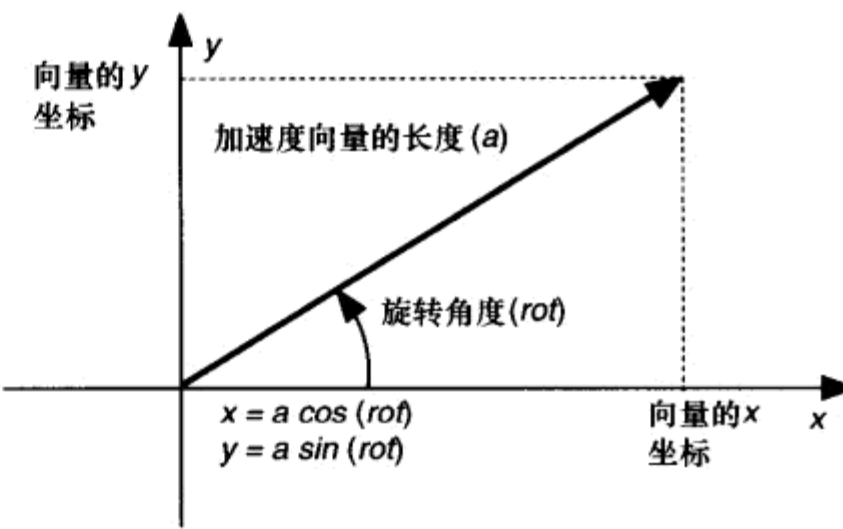


图 3.2.2 从极坐标系换算到笛卡尔平面直角坐标系

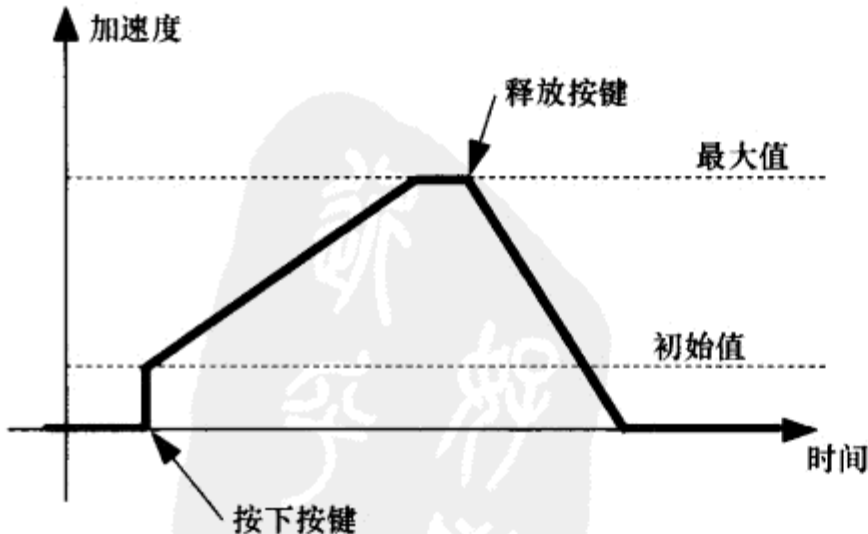


图 3.2.3 加速度/时间图

中所示的行为表现的是交通工具在加速按钮松开之后慢慢减速。

有的时候，可能为了避免和其他选手的车辆发生碰撞，会用很大的力气踩下刹车踏板。这时候，需要用到刹车按钮。它的使用很简单。当游戏者按下刹车按钮时，在方程式 3.2.4 中，不用当前加速度 a ，而用表示刹车的负加速度——即减速度（deceleration） b 。实际上， b 的值并不像在实际生活中那样代表刹车给车辆增加的摩擦力，而是相当于使用了倒车挡。在游戏中的最终效果是相似的。并且，刹车键还能当作倒车键来用，一举两得。

3.2.3 方向控制

手持游戏机上的按钮个数一般是很有限的。例如，任天堂 GameBoy Advance (GBA) 上只有两个标准按钮、两个扳机按钮（shoulder button），以及 4 个方向的十字键。在我们的实现中，切换武器及开关涡轮增压（turbo）已经用去了一部分按钮。如此一来，可以用来控制方向的就只剩下两个按钮了。幸好，两个按钮已经足够用了。

最简单的解决办法就是，让这两个按钮分别用来增加或减小汽车转动的角度。和加速按钮一样，方向控制按钮也只有按下和松开两种状态。也就是说，在每一帧中，转动角度（rot）都要作一定量（drot）的调整。

$$rot_n = rot_{n-1} - drot$$

(3.2.5)

假如把 $drot$ 设为一个固定值，比如 5 度，那么这个方向控制模型（steering model）看上去就会很不自然，因为汽车看上去是以一个固定的速度转动的，而实际生活当中，汽车转向装置是线性设备（转向快慢是由方向盘转动程度大小控制的）。要模拟这样的装置，就应该根据方向控制按钮被按住的时间长短改变转动速度。同加速踏板类似，我们要设置一些初始值以及最大值。此外，按钮按下以及松开后的速度改变率也应设为不同（见图 3.2.4）。

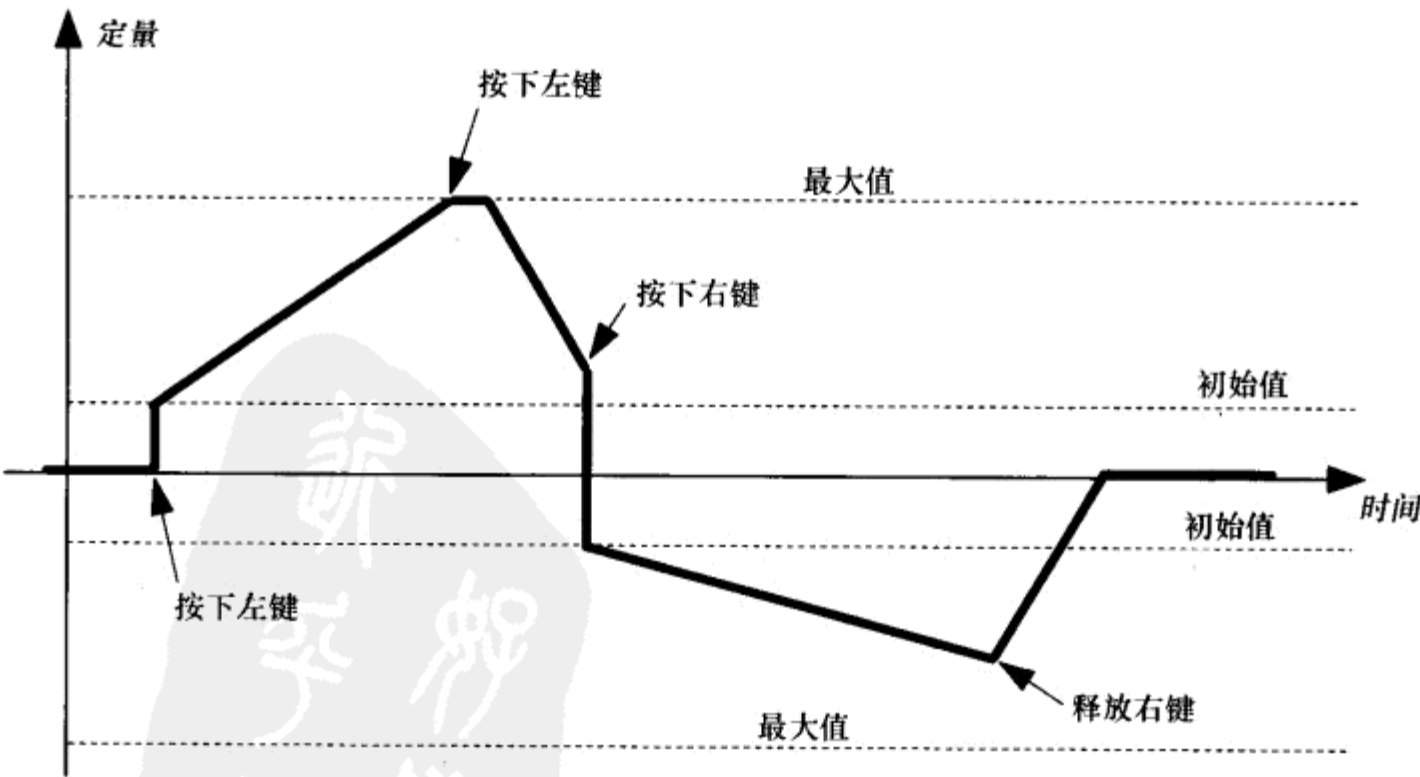


图 3.2.4 改变旋转的因子

在我们的游戏中，我们假设车辆可以自由转动。虽然这看上去有些不自然，但是它使游戏者可以更有效地控制车辆，尤其是在一些驾驶困难的地方。如果要模拟得更真实，可以使游戏者无法在车辆静止时转动汽车。此外，还可以把整辆车的转动基准点（pivot point）从车的中部移到后侧，以便模拟出前轮驱动控制方向的效果。

3.2.4 把所有要素结合起来

以上介绍的两个简单机制使游戏者能在一条设计好的赛道上驾驶车辆。由于速度和方向是分开控制的，汽车能做出一些有趣的特技效果，例如车轮打滑或用手刹使汽车转向 180 度。举个例子，在全速驾驶汽车的同时，按住其中一个方向键，你所驾驶的汽车就会向转向的另一侧甩尾（drift）。如图 3.2.5 所示，这样能做出一些很漂亮的赛车特有的动作。

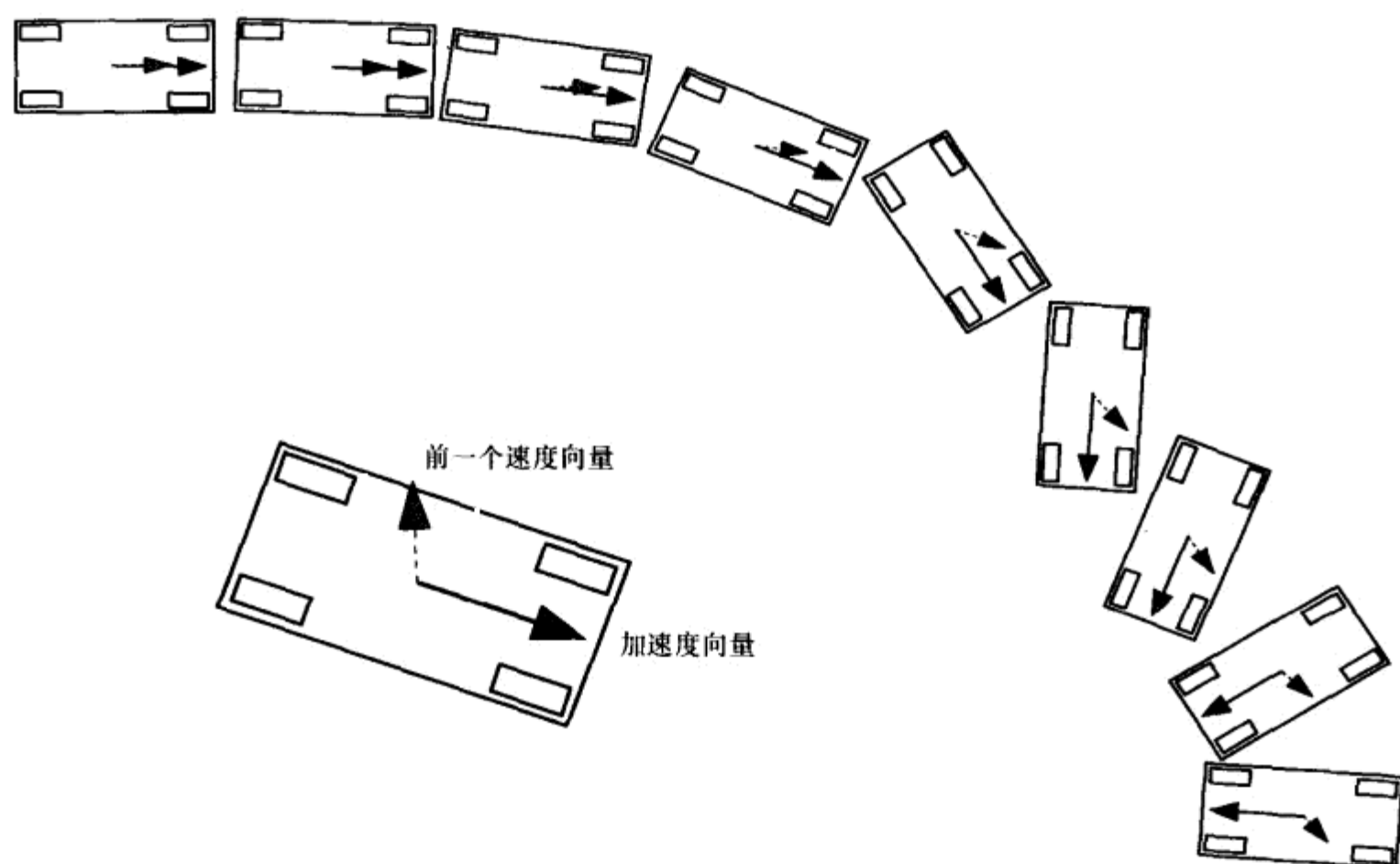


图 3.2.5 在连续几帧内的车辆旋转、加速度和速度矢量

3.2.5 地形的影响

在每一帧中，车辆都会探测它所在的地方属于哪一类地形。每一类地形有它自己的一套环境变量，影响着加速度、摩擦和车子的转动。在定义这些变量的时候，我们可以制造类似溅散了汽油的路面、结了冰的桥梁或烂泥地这样的效果，以显著地影响汽车驾驶。在模拟程序的开始处，插入类似于如下所示的代码片段（具体将会在下一节中详细介绍定点算法）来计算车辆的物理信息。

```
...  
// 计算地形的影响
```



```

// 我们用地形来改变加速度及转向参数，这里：
// accel = 最大加速度的大小
// friction = 摩擦力系数
// start_drot = 初始转向角度的数值
// max_drot = 最大的转向角度
// delta_drot = 增加的角度
// return_drot = 减少的角度

char terrain = getMask(car_pos_x, car_pos_y);

switch(terrain){
// 标准的路面
case TERR_ROAD :
    // 0.2 在 8 位定点运算中等于 51
    accel= 51;
    // 64225 等于 0.98 在 16 位定点运算中的大小
    friction    = 64225;
    // 0x100 == 1 == 正圆的 1/256
    start_drot = 0x100;
    // 0x500 == 正圆的 5/256
    max_drot = 0x500;
    // 0x80 == 正圆的 0.5/256
    delta_drot = 0x80;
    return_drot = 0x80;
    break;
// 冰面：我们的汽车会失去一些抓地力
case TERR_ICE :
    accel= 51;
    friction    = 65208; // 16 位定点浮数中的 0.995
    start_drot = 0x40;   // 0.25/256
    max_drot = 0x700;
    delta_drot = 0x20;
    return_drot = 0x20;
    break;
...

```

3.2.6 实现中遇到的问题

为了提高计算速度，并且减少在那些没有浮点运算处理器的平台上的开销，我们应该在计算中使用定点（fixed point，固定小数点）的方法。在定点数中，我们假设把小数点的位置都固定在一个位置，例如一个值的第 8 位后，请参见图 3.2.6。

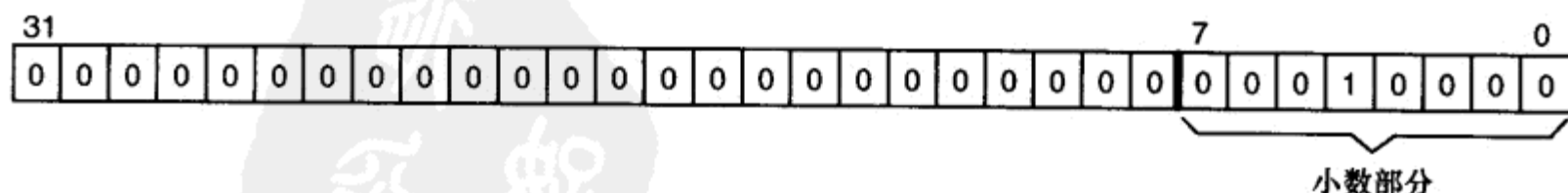


图 3.2.6 用 32 位整数表示的定点数，小数点后保留 8 个二进制位

这样的数值可以看成为被一个固定的除数相除。在这个例子中，除数的值是 $2^8 = 256$ ，

因此我们应当把值 (10000_2) 分析为 $16/256 = 0.0625$ 。

CPU 和编译器是不能直接处理定点值的, 只会把它们当成普通的整数变量。整数变量的操作十分快捷, 简直可以说加法和除法毫不费力。但是, 当处理两个定点数相乘的时候, 结果必须要作修正。乘出来的结果要适当调低 (比如说, 把它向右移)。右移的位数等于小数点所在的位数。例如:

$$\begin{array}{r} 0000\ 0000.0001\ 0000 = 16/256 = 0.0625 \\ \times \\ 0000\ 0000.0010\ 0000 = 32/256 = 0.125 \\ \hline = 0000\ 0010.0000\ 0000 = 512/256 = 2.0 \end{array}$$

如你所见, 把 0.0625 和 0.125 相乘却得到结果 2.0, 这显然是错误的。正确的结果应该是向右移 8 位, 得出 0.0000 0010, 也就是等于 $2/256 = 0.0078125$, 这下就对了, 与 $16/256 \times 32/256 = 0.0625 \times 0.125$ 完全相符。

定点数和整数相乘的结果不需要修正, 但是定点数和整数相加或相除则需要先把整数放大 (左移) 成定点数的格式。要把定点数换算成整数值, 只要把定点数往右移 8 位, 丢弃小数部分即可。

对于除法运算, 我们决定不采用运行时运算的方法。你也可以这样做, 不过说到底这还是取决于你可支配的 CPU 处理时间的多少。我们认为用一张大的除法表比用 CPU 去除两个数好。虽然由于这样处理, 精确度会有所降低, 但是计算过程能以最快速度进行。

同样, 三角函数 (正弦、余弦等) 的计算也使用 32 位 (定点精度 16 位) 的计算表来表示, 单位是 $1/256$ 个圆周角 (即 360 度或 2π 的 $1/360$)。你可以创建如下的宏来进行查表:

```
#define _SIN16(a) (sincos16[(a)&0xff])
#define _COS16(a) (sincos16[((a)&0xff)+64])
```

请看, 余弦的宏虽然和正弦的宏使用同一张表格, 但是把索引值位移了 64 个单位角度之后才进行查找 ($64/256$ 恰好表示直角 90 度)。这使我们节约了宝贵的内存资源。

在游戏引擎中有那么一些参数 (例如摩擦系数), 即使很小的变化也会导致它们的改变。因此, 我们需要把它们表示为 16 位的定点数。这样表示的话, 进行放大时要更谨慎一些, 但是总的原理还是一样没变。

在开发过程中我们发现, 当把小的负定点数和正数相乘时会出现一些奇怪的现象。有时候在相乘之后, 即使乘数不为 1, 得出的结果还是与被乘数相同。造成这种现象的元凶就是计算结果的溢出。为了避免这种情况, 办法之一就是检查乘数是否是负数。如果是, 计算前首先对它求反, 然后再用求反后的结果进行计算。

```
if (value < 0)
    value = -(((value) * any_factor)>8);
else
    value = (value * any_factor)>8;
```

3.2.7 可以改进的地方

如果你想在自己的游戏中使用这个引擎, 并提高它的精度, 那么可以考虑对加速器或方

向盘的线性模拟加以改进。不妨使用一个查询表，用这表定义在持续按下按键一段时间后加速踏板和方向盘的行为。你可以使用大小为 64 或者 128 的 16 位（8 位小数部分）定点数组，来存储根据按住或释放相应按键得到的 64 或 128 帧的加减速参数。例如：你可以用图 3.2.7 中所示的数值来创建你的数组，以模拟这样一个效果：一开始时加速度很猛，随后加速度慢慢变缓，最后是涡轮压缩机延迟启动，加速度进一步加大。

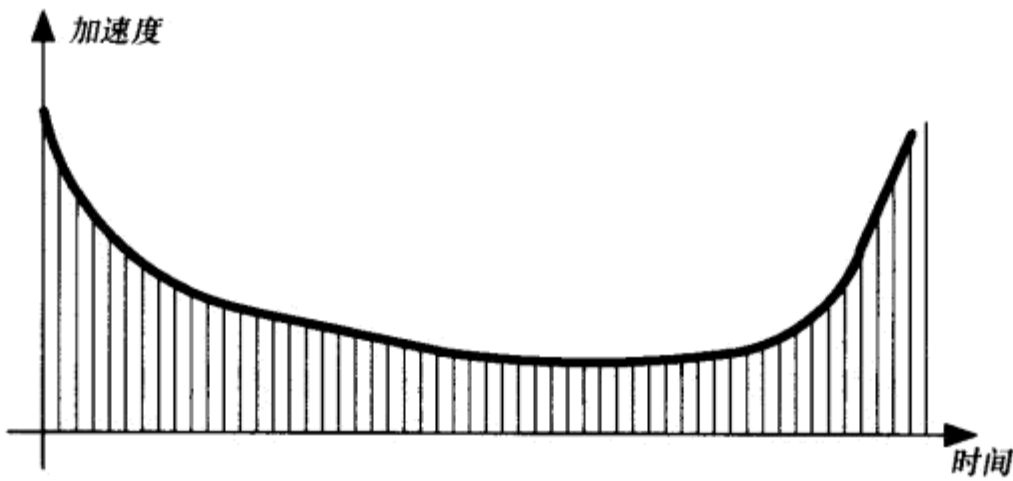


图 3.2.7 模拟复杂的加速度改变

3.2.8 结论

这里介绍的算法和代码片段都只是组成整个骨架的基础工作，要完全做出读者想做的东西，还需要进一步更详尽更精确的调节（Tune）。为了在商业作品中使用这套系统，必须开发出相应的工具软件来准确地调整一些交通工具及地形的参数。一旦这些工具软件到位，就可以用来创造出各种类型的游戏，除了竞速类游戏外，还可以是射击类游戏、platform 游戏，甚至是任何涉及到按照物理定律运动的物体的二维游戏。



3.3 编写基于 Verlet 积分方程的物理引擎

作者: Nick Porcino, LucasArts

E-mail: nporcino@lucasarts.com

译者: 李鸣渤

审校: 沙鹰

游戏中常常需要建造一些运动的物理过程模型, 例如天气、木箱、交通工具、机械系统、人物等。如果这些过程可以通过方程式来建模的话, 就可以得出一套算法来模拟这些过程。游戏中让人特别感兴趣的一类系统是刚体动力学。一个具有物理属性的刚体系统模型模拟器可以互相接触、碰撞。同样也可以模拟像布料、头发等柔体。每帧中, 通过在系统中输入外力, 可以使模拟变得格外有趣。



ON THE CD

实现一个相当准确和复杂的物理引擎其实比大家想象的要简单得多。这篇文章将通过介绍一个简单的引擎来阐述这个问题。随书附送的光盘包含了这个引擎的部分代码。

3.3.1 关于物理引擎

模拟分为以下几种类型: 离线模拟 (offline), 模拟过程比实际事物发生过程要慢; 在线模拟 (online), 模拟过程和实际事物发生过程同步; 交互模拟 (interactive), 运行速度非常快, 在一个循环周期中, 使用者可以在模拟的同时与系统发生互动; 实时模拟 (real-time), 它保证系统能以一定的帧速率更新。下面展示一个没有很多物体的交互模拟 (interactive) 物理引擎。

一个游戏除了物理引擎之外, 还包括许多其他部分, 比如渲染、文件系统、人工智能, 等等。一个物理引擎不能实现上述的任何部分, 而且也不应该与它们有任何直接联系。也就是说, 人工智能系统或者渲染系统可以通过物理引擎得到物体的变换矩阵等信息, 但物理引擎本身并不提供人工智能或渲染的功能。

大部分游戏物理引擎的参考资料要么重点讲述非刚体(如布料和布偶)模型的 Verlet 积分器, 如弹簧系统 [Jakobsen03], 要么着重阐述刚体的普通微分方程 [Hecker96]、[Baraff97] 的求解。这篇文章结合转动效果 (angular effect) 的运用, 阐明了 Verlet 积分器也可以应用到刚体的模拟中去。此外, 大部分物理引擎关于碰撞的阐述主要使用约束器的方法 [Smith04], 或者

是惩罚性系统 (penalty system) [Jakobsen03]。而这篇文章中介绍的引擎使用了相对较新的技术——基于冲量 (impulse-based) 的动力模拟以及微碰撞 [Mirtich95]。

本篇文章将不包括对约束器系统、如何模拟有关节的身体, 以及运动方程式的介绍。要了解这些基础知识, 请参照 Chris Hecker [Hecker96] 或 David Baraff 的很多优秀文章。[Smith04] 包含了对雅可比约束器的讨论。

3.3.2 刚体

刚体 (rigid body) 物体同时兼具动态和静态的属性。动态属性是指它的位置和速度, 以及方向、角速度 (angular velocity) 和角动量 (angular momentum); 静态属性是指它的大小形状、质量、惯性张量 (inertia tensor)、(摩擦的) 速度阻尼 (velocity damping), 以及用于计算碰撞的信息。惯性张量 (inertia tensor) 是物体体积和质量的三重积分, 当物体质量或形状有所改变时, 要重新计算惯性张量。刚性物体可以跟踪, 如物体能否旋转, 在碰撞中表现如何、是否活跃等其他有用信息。

在物体新的动态状态建立时, 需要追踪前一步长结束时的动态状态。每一帧内, 作用于物体的力和力矩 (包括摩擦力) 都累加进入积分器中, 用作积分。

3.3.3 积分器

积分器 (integrator) 有两大类: 隐性 (implicit) 和显性 (explicit)。隐性积分器要求解一系列的方程式来计算当前的步长, 而显性积分器则利用显性有限差分进行迭代。显性欧拉积分器是所有积分器中最简单的。Verlet 积分器是一个基于二阶显性微分的积分器, 这个积分器在泰勒运动方程式的基础上展开两步, 一步向前, 一步向后得到的。

$$x(t+d) = x(t) + v(t)dt + F(t)dt^2 / 2m + \dots \quad (3.3.1)$$

$$x(t-d) = x(t) - v(t)dt + F(t)dt^2 / 2m + \dots \quad (3.3.2)$$

把 (3.3.1) 和 (3.3.2) 相加得出游戏引擎中普遍使用的基本 Verlet 积分器。

$$x(t+dt) = 2x(t) - x(t-dt) + F(t)dt^2 / m + O(dt^4) \quad (3.3.3)$$

方程式 3.3.3 只需要知道物体的位置; 每一步长的速度可以大致算出。这种积分器和准确测出速度的积分器相比, 所占内存要少。在无法可靠测算速度的情况下, 它也占优。这种积分器尤其适合计算成本高的模拟, 比如对布料及有限要素系统 (finite element system) 的模拟。方程式 3.3.3 的主要缺点是, 由于力产生的加速度是位置和速度的直接相加, 积分器很快就会从准确的结果发散。

在许多引擎中, 积分器的发散度是通过强阻尼来减小的, 这对模拟的影响比较大, 特别是系统会迅速地流失能量, 因而模拟会显得迟钝松散。在布料模拟等其他应用中, 运动阻尼是可以接受的, 因而计算成本低廉的基本 Verlet 积分器就成为了首选。在涉及刚体动力学的

模拟中，更适合选用其他安全稳定的积分器。

Verlet 积分器有两种最常见的形式：蛙跳（leap frog）积分器和基于速度的 Verlet 积分器。蛙跳积分器对单个的积分步骤进行分解，求出在计算速度中点的位置以及在计算位置中点的速度。蛙跳积分器具有较高的数值准确性，但是速度仍然是推算出来的而不是明确积分得出来的。基于速度的 Verlet 积分器则能确切地求出每一步的位置及速度。虽然它因存储速度而增加了成本，但它是所有 Verlet 积分器中最准确的。

基于速度的 Verlet 积分器的构成使人回想起中学物理。

$$x(t + dt) = x(t) + v(t)dt + a(t)dt^2 / 2 \tag{3.3.4}$$

在这个方程式中， x 是位置， t 是时间， dt 是时长， v 是速度， a 是加速度。加速度是由在每帧中输入的所有外力的总和除以物体的质量推算出来的。

$$a = F / m \tag{3.3.5}$$

这里的力可以来自弹簧、发动机、力场、重力、摩擦、粘力，等等。

$$F = F_{external} - \sum_{i=springs} d_i k_i x_i \tag{3.3.6}$$

这里， $F_{external}$ 表示除了弹簧以外的外界所作用的力，而方程式的其余部分是对弹簧的描述。弹簧单位长度的当前方向 d ，计算为指向与弹簧相连接物体的相反方向； k 是弹簧的弹性系数； x 是弹簧的当前长度与原长度之差。假如弹簧被拉伸，物体就会被拉向弹簧方向；假如弹簧被压缩，物体则会被推向相反方向。在一步长时间内弹簧长度改变时，阻尼因子会通过每个弹簧的阻尼固定的次率从公式 3.3.6 中减去。

在现实中，积分步骤比简单的中学物理要复杂得多：速度更新方程式包含了在中点及每一个步长结束时加速度的计算。

$$v(t + dt) = v(t) + [a(t) + a(t + dt)]dt / 2 \tag{3.3.7}$$

如果对象是布料之类的柔性物体，则积分器的工作到此结束。但是如果对象是刚体，还要对角度的影响进行积分。角速度我们用角动量推算得出，因为对角动量的积分远比对角速度的积分要简单 [Hecker96]。要计算新的时长的角动量很简单，只需把力矩和时长的乘积与前一个时长的角动量相加即可。

$$L(t + dt) = L(t) + [T(t) + T(t + dt)]dt / 2 \tag{3.3.8}$$

这里 L 是角动量， T 是力矩。和方程式 3.3.6 中力的计算一样，弹簧也应作为力矩的一部分考虑在内。弹簧的接触点用来形成适当的力矩，如图 3.3.1 所示。

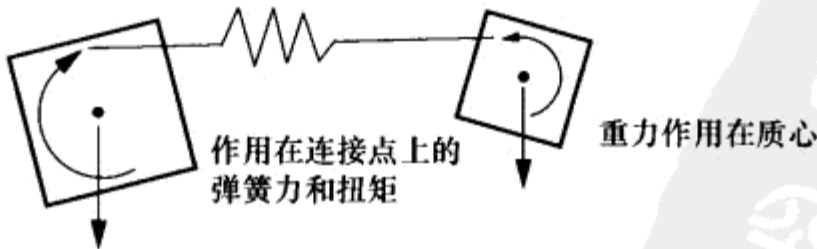


图 3.3.1 没有连在物体质心的弹簧产生的扭矩

要提醒的一点是，在已知物体上有一个力的作用点的情况下，力矩是通过力作用点到

质心的距离与力的叉积计算得出的。关于物理模拟中弹簧系统的精确分析,请参见[Kačić03]。角速度的计算如下:

$$\omega = (RIR^T)^{-1} L / m \quad (3.3.9)$$



其中, ω 是角速度, R 是旋转矩阵, I 是惯性张量, L 是角动量, m 是质量。括号中的矩阵乘法把惯性张量从世界空间 (world space) 移向物体空间 (body space)。矩阵的求逆操作可以在建造了刚体数据结构时通过预先计算质量的倒数消掉。质量也可以在这个时候被除掉。另一个可以进一步优化的地方就是, 认识到在动力学上, 对称的物体存在一个只在对角线有值的张量矩阵, 并且每个球形物体在整个对角线上的值都是一样的。如果那些情况在积分器中有专门描述, 角旋转效果的计算就可以大大简化。随书附送的光盘的样本代码对四元数进行了进一步地详细介绍。

3.3.4 物理引擎

这篇文章中介绍的基于 Verlet 的物理引擎并不完全支持一般性刚体。下面介绍对它的几种简化, 以增加系统的稳定, 降低系统资源的使用, 加快计算速度。

- 要简化角旋转效果的计算, 可将物体的位置设在质心, 而不是几何中心。
- 没有必要保存一套完整的惯性张量, 取而代之的是, 引擎仅仅支持形状对称的物体 (如箱子、球体、圆环), 其惯性张量是对角矩阵。这样可以简化积分器。
- 摩擦被简单看作与运动方向相反的作用力。有一点奇怪的是, Coulomb 摩擦力是一阶既独立于速度又独立于接触面的函数, 在 Coulomb 的模型中, 摩擦力只是表面法向作用力的函数。



就随书附送的光盘中的源代码而言, 物理引擎 (PhysicsEngine.h) 可以提供刚体的接口界面、运行模拟、解决碰撞。通过它, 可以得到模拟步长后每一个刚体的变换矩阵。一个刚体的位置、速度、方向对使用者来说都是一目了然的, 而加速度、角速度和角加速度 (angular acceleration) 则不然, 因为那些不能明确看出的特性是由引擎通过每帧中输入的初始条件、力和力矩推算出来的。这些数值可以提取出来, 但实际操作起来, 要用的时候并不多。图 3.3.2 阐明了用于使用物理引擎的各类之间的关系。

PhysicsEngine 类是物理引擎的主要界面; PEAux 是私有的实现体。RigidBody 有两个 DynamicState 对像和一个 RigidAccumulator。RigidBody 的静态特性, 如质量, 是作为 RigidBody 成员来被保存。要注意的是, 从 PhysicsEngine 并不能直接得到 RigidBody 的对象指针, 但是在创建了刚体后能返回惟一的标识符。而游戏必须追踪所有返回的惟一标识符。这些标识符会用于建立及获得 RigidBody 的特性, 找回它的变换矩阵。这一方法是从软件管理设计模式出发的。

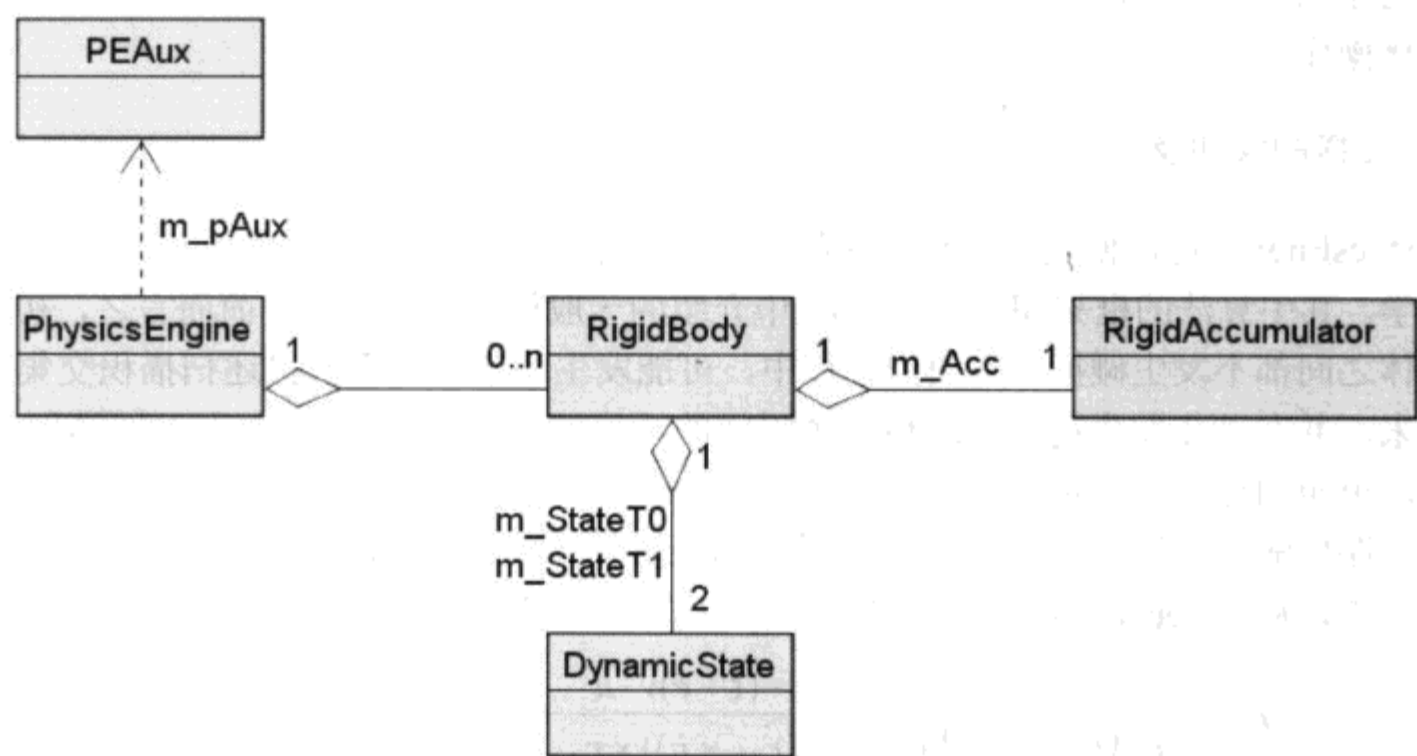


图 3.3.2 物理引擎类

1. 模拟循环

在模拟中加入并初始化了所有刚性物体后，每走一帧，使用者的主循环就要使用一次 **PhysicsEngine** 的 `Simulate()`方法，把上次运行后失去的时间补回来。主循环应根据最大值对其进行调整，提供从上次调用后所经历的时间，如 1/20 秒，以使系统保持稳定。同样，虽然像 1/120 这样很小的数字可以被当作时长，而零却不能。稳定标准不是寥寥数语能说清楚的，因此在此不予讨论，相关的信息可以参考 [Kačić03]。

创建刚性物体，初始化 T1 状态
模拟

- 把之前的 T1 状态拷贝到 T0 状态
- 速度 += (a * dt) / 2, 方程式 3.3.7
- 角动量 += (T * dt) / 2, 方程式 3.3.8
- 计算位置, 方程式 3.3.4
- 计算角速度, 方程式 3.3.9
- 计算力, 方程式 3.3.6
- 计算力矩, T, 图 3.3.1
- 更新加速度, a, 方程式 3.3.5
- 速度 += (a * dt) / 2, 方程式 3.3.7
- 角动量 += (T * dt) / 2, 方程式 3.3.8
- 碰撞及解决

标准化

模拟这一步骤非常明白易懂。首先，准备好所有内部状态，以做模拟。在用 `Simulate()`方法积分之前，要预先执行一步，就是把当前的数值复制到之前的数值、把新数值复制到当前的数值，依此类推。然后，运动的方程式才被积分。要注意的是，方程式 3.3.7 和 3.3.8 被分开了，这样外力可以平均地加在积分的中点。如果在过程中实施了一个约束器，那么这个约束器应当作用在中点上，碰撞（一类非特殊的约束器）被检测分解。由于误差的积累和其

他类似的原因，一些矩阵会变为非正交矩阵，为了修正这样的错误，有必要对内部状态值进行标准化操作。

2. 碰撞的侦测及分解

PhysicsEngine 可以侦测及分解碰撞问题。为使思路清晰，下面要介绍的版本只涉及基本的几何学。其中算法的框架和 [Nettle00] 中介绍的一般框架是一样的。简而言之，初始状态下各物体之间都不发生碰撞。在模拟过程中，可能发生的碰撞由解析描述扫描积交集的方程侦测出来，并在动作发生前通过运用一个瞬间的冲量解决 [Mirtich95]。由于扫描积 (swept volume, 也叫扫掠体) 决定了具体接触点和接触时间，不会产生非法状况，因此也就不需要特别的试探性解决方法了 (如，在运动路径上的逆向追溯直至碰撞不再发生)。控制碰撞解决的方程式就是从 [Hecker96] 第三部分演化而来的对冲量的计算。

$$j = \frac{-(1+e)v \cdot n}{n \cdot n(M_A^{-1} + M_B^{-1}) + [(I_A^{-1}(r_{AP} \times n)) \times r_{AP} + (I_B^{-1}(r_{BP} \times n)) \times r_{BP}] \cdot n} \quad (3.3.10)$$

其中 j 是碰撞产生的冲量， e 是复原系数 (coefficient of restitution)， v 是两个物体间的相对速度， n 是碰撞法线 (collision normal)， M 是 A 物体和 B 物体的质量， I 是惯性张量，而 r 是从 A 物体和 B 物体的质量中心到碰撞点 P 的向量。

由于这个物理引擎依赖的解决方案是基于冲量的，引擎不需要在静态摩擦力和动态摩擦力中进行转换。引擎把所有碰撞视为动态碰撞，并通过微碰撞 (micro-collision) 来处理滑动触点 [Mirtich95]。这减少了复原力的计算。所有接触的模式，包括持续接触，不管是静止的、滑动的，还是滚动的，都被当作作用在物体上的一系列微小冲量来处理。在这种基于冲量的模拟中，即使是静止的物体也会与支持面发生微小而迅速地接触。解决这些碰撞只用了由接触点得到的信息。发生微碰撞时物体的速度变得足够小，在这种情况下，物理引擎让物体处于休眠状态，不再对它们进行积分，直至又有其他物体或外力参与进来。

在实际中，微碰撞技术集中在更传统的动态/静态模型上 [Mirtich95]。它在数字上更稳定更完善，因为在不同的摩擦力模式中转换的时候没有发生中断。所有物体在静止的时候都处于休眠状态。因此在模拟一个堆满木箱的房间时，木箱的数目可以成百上千，因为只要它们不动，就根本不占用 CPU 任何时间。

3. 碰撞处理及游戏逻辑

物理引擎运行的时候要知道两个物体什么时候发生碰撞。为了达到这个目的，有一组回叫信号接口。回叫信号可以提供一些有趣的数值，在运行时给系统提供信息，例如碰撞是否足够猛烈以致于打碎物体。回叫信号也会返回数值，告诉系统需不需要对碰撞做出反应。实际操起来它作用很大，因为有些碰撞可能是有条件的，例如，某种特殊的血液也许能毫无障碍地穿透木头，却穿透不了钢铁。

PhysicsEngine 类在模拟的步骤中，会通过回叫信号报告所有碰撞。系统会以任何被认为恰当的方法处理碰撞，但以下这一情况例外：为了避免非法状态的发生，系统不能调用任何 PhysicsEngine 方法，最后系统在模拟过程发生后，通过调用 Get 函数来得到每一刚体的变换矩阵及其他一些有用信息。

4. 复杂几何——概述

如果 PhysicsEngine 的运用从简单的几何形体延伸到更为复杂的几何形体, 或者飞机、箱子的话, 建议使用像 [Kačič03] 中介绍的惩罚性碰撞解决方法。在这样一个系统中, 在所有接触点或物体穿透点都使用长度为零、硬度达到最大的弹簧, 并且在碰撞处理达到一个可以接受的程度以前, 系统都以小步长积分。解决碰撞后, 这些弹簧也就不用了。弹簧最大硬度的计算和积分器的稳定性相关, 具体内容请读者参考 [Kačič03]。基于拉格朗日乘法器 (Lagrange-multiplier) 的技术也非常适用, 请参考 [Smith04]。

3.3.5 针对特定平台的考虑

如果可能的话, 向量和四元数对象可以以数值, 而非指针传递。有许多 Vector3 和 Quaternion 的应用程序是 SIMD: x86 处理器的 SSE、PS2 的 VU0、PPC 的 AltiVec。数值的传递使对象传递可以在一个 SIMD 寄存器中进行。示例代码忽略了这个优化过程, 只简单地使用了一个实数数组。分解过程揭示了现在的微软和 CodeWarrior 编译器中用指针器或引用 (Reference) 传递 SIMD 数值, 太多情况下调用了不必要的装载和存储过程, 而通过数值传递则可能使得出的代码更为快速, 也更可能优化各项功能之间的运行。即便如此, 检查通过数值传递的常量对象是否创建了临时变量还是很重要的, 因为在这种情况下, 根据版本的差异, 编译器的表现会有所波动。

3.3.6 扩展引擎的功能

筛选 (culling) 是物理引擎很重要的一方面。而基于入口的游戏引擎则是其中要处理的一个典型问题。我们不需要对非活动的入口区作物理描述; 我们可以为每一个区域作一个特别的 PhysicsEngine 实例。根据这点, 物体 (如游戏者扮演的人物) 有时会在各区域之间移动, 那么也就是在各个 PhysicsEngine 之间移动了。我们在把物体从一个引擎撤走, 再放入另一个的时候, 为避免出现不连贯, 一定要小心。

也要适当对 LOD (Level Of Detail) 予以重视。我们可以把许多很棒的技巧运用在这里。当物体垂直的速度大约为零的时候 (如冰球在冰面上短暂滑行), 可以不考虑重力, 以减少大量的分解碰撞计算 (检测仍然要做, 但是由于物体不会被重力推入地下, 这类碰撞不必给予分解)。我们可以把碰撞检测的方法恢复到球体或者平面检测法, 这样准确度虽然降低但速度提高了。我们还可以完全关闭碰撞检测、关闭角积分器、放宽物体置于睡眠状态的标准, 以及保持物体的休眠状态。在以上所有情况中, 一旦 LOD 更改为更高的水平, 物体就应立即进入有效 (valid) 的状态及位置。

开发 PhysicsEngine 的下一步是加入一个约束器系统 (constraint system)。这是个很大的主题, 有关的文章都可以申请版权了。关于弹簧质量系统以及相关的主题, 在 [Jakobsen03] 中作了很好的阐述; 关于约束器系统, 在 [Smith04] 中进行了深入研究。

运用复杂的碰撞处理系统, 如 OPCODE [Terdiman00], 可以大大改进碰撞系统并提高通用性。在这一领域, [Moravánszky04] 讨论了如何减少接触以加速处理。

与动画引擎的结合是一个很吸引人的延伸,而且能以此取得布偶效果及二级动画。在播放动画帧中的某一时刻,动画引擎被关掉,物理引擎取而代之。这时,有些变量需要用来做物理计算。速度很容易推算出来,只要知道前面两帧动画及时长;但是角动量则是比较大的挑战,而且超出了这篇文章的讨论范围。首先要使用物理引擎已经认识的基本图形,如圆柱体及立方体,对一个物体的相关部分建模。详细情况请参考 [Baraff97]。

3.3.7 结论

这篇文章介绍了一个物理引擎,它使用了一些很有趣的技术。文中描述并改进了基于速度的 Verlet 积分器,讲述其如何运用在游戏中,它对模拟刚体和柔体动力学都适用。文章还描述了基于冲量的包含微碰撞的动态仿真模拟。最主要的是,作者展示了如何解决许多常见并且容易被疏忽的问题,以实例说明了即使是引擎中包含一些难度很高的因素,如角运动积分,编写这样一个引擎其实也是很简单的。作者希望通过这篇文章能揭开一些围绕物理引擎的谜团。

3.3.8 参考文献

[Baraff97] Baraff, David, "Physically Based Modeling, Principles and Practice, Online SIGGRAPH '97 Course Notes," available online at www-2.cs.cmu.edu/~baraff/sigcourse/index.html, 1997.

[Hecker96] Hecker, Chris, *Game Developer Magazine Physics Series*, available online at www.d6.com/users/checker/dynamics.htm, October 1996–June 1997.

[Jakobsen03] Jakobsen, Thomas, "Advanced Character Physics," available online at www.gamasutra.com/resource_guide/20030121/jacobson_01.shtml, January 21, 2003.

[Kačić03] Kačić-Alesić, Zoran, Marcus Nordenstam, and David Bullock, "A Practical Dynamics System," *Eurographics/SIGGRAPH Symposium on Computer Animation* (2003).

[Mirtich95] Mirtich, Brian, and John Canny, "Impulse-based Simulation of Rigid Bodies," in *Proceedings of 1995 Symposium on Interactive 3D Graphics*, April 1995, available online at www.cs.berkeley.edu/~jfc/mirtich/impulse.html.

[Moravánszky04] Moravánszky, Ádám, and Pierre Terdiman, "Fast Contact Reduction for Dynamics Simulations," *Game Programming Gems 4*, 2004.

[Nettle00] Nettle, Paul, "Generic Collision Detection for Games Using Ellipsoids," available online at www.fluidstudios.com/publications.html, October 5, 2000.

[Smith04] Smith, Russell, "Constraints in Rigid Body Dynamics," *Game Programming Gems 4*, 2004.

[Terdiman00] Terdiman, Pierre, "OPCODE, Optimized Collision Detection," available online at www.codercorner.com/Opcode.htm, 2000.

[Verlet67] Verlet, L. "Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules," *Phys. Rev.*, 159, 98–103, 1967.

3.4 刚体动力学中的约束器

作者: Russ Smith, Open Dynamics Engine (www.ode.org) 的作者

E-mail: russ@q12.org

译者: 李鸣渤

审校: 许竹钧

现代电脑游戏使游戏者身处的虚拟世界越来越逼真了。真实性增加的原因有些是因为复杂的渲染, 有些来自于人工智能, 还有些则是对游戏对象动作更详细的模拟。汽车能像真实世界中一样刹车、翻筋斗; 桥在有人走过的时候要能摇摆; 砖墙在受到碰撞的时候要能碎裂; 游戏中的人物掉下去的时候要显得真实。

创建这些动作的一种广泛流行使用的技术就是刚体模拟。有关刚体模拟的书籍往往会介绍标准的关节(joint)种类, 以供使用者把身体的各部分连接起来, 例如铰链(hinge)以及球窝关节(ball-and-socket joint)。另一些书籍阐述了使用者如何创建自己的关节类型。掌握了这一点能给模拟新的环境带来不少好处。例如, 一种新的关节类型能使过山车的车体与轨道连在一起, 或者能建立模型模拟一种特别的悬浮模式, 使车轮和底盘连在一起。

基于笛卡尔坐标系/拉格朗日乘法器的刚体模拟(如[Baraff96])是一种很流行的技术, 它使用了速度约束器(velocity constraint)。这篇文章就要阐述如何用简单的方法为这种模拟建造新的关节。文章针对 Open Dynamic Engine (ODE), 一个开源的刚体模拟库[Smith03]作了详细介绍。其他的模拟技术, 如坐标系消减法(reduced coordinate method)[McMillan94], 文章没有涉及。读者将会看到, 无需高深的数学, 如何利用简单的零件建造出关节。这篇文章的前提是读者对刚体模拟的基本概念已经熟悉, 否则, 可以先参阅有关这方面的入门书籍, 如[Witkin97]和[Hecker96]。

3.4.1 基本要点

在刚体模拟中, 每个物体的运动都由 6 个自由度控制(3 个用作位置平移, 3 个用作方向旋转)。约束器(constraint)使物体丧失了系统中的自由度——这些是数学上的约束, 约束物体之间的相对移动。约束器是由模拟器强制产生的, 如通过把自动计算的约束力作用在每个刚体上。约束器通常以连接成对物体的关节的形式存在。在本文中, 约束器和关节表示相同的意思。

1. 速度及加速度

我们把一个物体的线性速度及角速度分别定义为向量 $[v_{ix} \ v_{iy} \ v_{iz}]$ 和 $[\omega_{ix} \ \omega_{iy} \ \omega_{iz}]$ 。线性速度是在物体的参照点（Point Of Reference，简称 POR）上进行测量的，而角速度是沿着这个参照点的旋转测量的。向量 \mathbf{p}_i 给出了 POR 的位置。通常，一个物体的 POR 就是其质心。每个物体也有线性加速度及角加速度的向量，也就是速度向量在时间上的导数。为了方便，我们把这些值归为 6 向量（注意：该符号表示 \mathbf{x} 在时间上的导数）。

$$\begin{aligned} \text{物体 } i \text{ 的速度: } \mathbf{v}_i &= [\dot{\mathbf{p}}_i \quad \dot{\boldsymbol{\Omega}}_i]^T = [v_{ix} \ v_{iy} \ v_{iz} \ \omega_{ix} \ \omega_{iy} \ \omega_{iz}]^T \\ \text{物体 } i \text{ 的加速度: } \mathbf{a}_i &= [\ddot{\mathbf{p}}_i \quad \ddot{\boldsymbol{\Omega}}_i]^T = [\dot{v}_{ix} \ \dot{v}_{iy} \ \dot{v}_{iz} \ \dot{\omega}_{ix} \ \dot{\omega}_{iy} \ \dot{\omega}_{iz}]^T \end{aligned} \tag{3.4.1}$$

2. 运动方程

一个物体 i 在牛顿定律 ($\mathbf{f}_i = \mathbf{M}_i \mathbf{a}_i$) 的作用下运动，其中， \mathbf{M}_i 是质量矩阵， \mathbf{f}_i 是作用在物体上的力及力矩。（下面所说的力，均包括力及力矩）。我们一般说： $\mathbf{f}_i = \mathbf{f}_{iv} + \mathbf{f}_{ie} + \mathbf{f}_{ic}$ ，其中 \mathbf{f}_{ie} 是非对称旋转惯力（nonsymmetric rotational inertia）的旋转力矩（rotating torque）， \mathbf{f}_{ie} 指一些由“外部”作用的（或者说由使用者作用的）力， \mathbf{f}_{ic} 是约束力。 \mathbf{f}_{ic} 在每个时长都由模拟器计算，这样刚体运动才会满足约束器。

3. 速度约束器

单个物体的速度约束器就是简单的说明了物体的速度向量可以在哪些范围区间，不能在另一些范围区间。单个物体约束器的例子之一是把物体和空间中一个固定点通过铰链相联结。单个物体约束器是由矩阵方程式 $\mathbf{J}_1 \mathbf{v}_1 = \mathbf{c}$ 表达的。展开一下，就变成了：

$$\begin{bmatrix} J_{11} & J_{12} & J_{13} & J_{14} & J_{15} & J_{16} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ J_{m1} & J_{m2} & J_{m3} & J_{m4} & J_{m5} & J_{m6} \end{bmatrix} \mathbf{v}_1 = \begin{bmatrix} c_1 \\ \vdots \\ c_m \end{bmatrix} \tag{3.4.2}$$

约束器的巧妙之处就在于模拟器会通过计算 \mathbf{f}_{ic} ，使 \mathbf{v}_1 总是满足这条方程式。被称为雅可比约束器的 \mathbf{J} 有 m 行，其中 m 是约束器的阶，或者是从系统中拿掉的自由度的度数。 \mathbf{J} 和 \mathbf{c} 里面的值在每个时长中都作典型的变化；这是因为这些值总是由物体精确的位置和方向来决定的。两个物体的约束器有类似的方程式： $\mathbf{J}_1 \mathbf{v}_1 + \mathbf{J}_2 \mathbf{v}_2 = \mathbf{c}$ ，其中， \mathbf{v}_1 和 \mathbf{v}_2 是两个连在一起的物体的速度。把三个或以上物体放在同一个方程式的约束器是可能存在的，但是这很少起作用，而且 ODE 也不支持。

3.4.2 约束器构造模块

约束器代码计算每个时长的 \mathbf{J} 和 \mathbf{c} 。 \mathbf{J} 和 \mathbf{c} 可以一行一行地设计，其中，为了控制沿一条特定轴的速度，每一行独立运作。下面的例子从很简单的单个物体约束器开始，然后把它们组合起来形成更复杂的约束器。

约束器一：沿 z 方向上没有运动

(POR 在 xy 平面上)

考虑一单个物体约束器 (见图 3.4.1 #1):

$$[0 \ 0 \ 1 \ 0 \ 0 \ 0]v_1 = 0 \quad (3.4.3)$$

通过乘法展开给出 v_{1z} , 因此这个约束器把物体在 z 方向的 POR 速度设为零。另一种解释就是约束物体的 POR 只能在 xy 平面移动。注意, 这个约束器并没有说到 z 位置究竟是什么, 因为被约束的只是速度, 而非位置。

约束器二：在 z 上没有旋转

考虑约束器 (见图 3.4.1 #2):

$$[0 \ 0 \ 0 \ 0 \ 0 \ 1]v_1 = 0 \quad (3.4.4)$$

这里所说的是 $\omega_{1z}=0$ 。因此, 这个约束器使物体不能绕着 z 轴转动。但是它仍然能绕着与 z 轴垂直的轴转动。

约束器三：任意平面上的 POR

$$[a_x \ a_y \ a_z \ 0 \ 0 \ 0]v_1 = 0 \quad (3.4.5)$$

这个约束器 (见图 3.4.1 #3) 除了在沿向量 $[a_x \ a_y \ a_z]$ 的 POR 速度为零外, 其他的与约束器一相似。另一种解释就是, POR 运动被约束在法向量 $[a_x \ a_y \ a_z]$ 的平面上。向量 \mathbf{a} 不一定是一个单位向量。

约束器四：围绕任意一条轴没有旋转

与约束器三类似, 以下这个约束器 (见图 3.4.1 #4) 约束转动不绕着 q 轴发生, 只绕着在那些在法线为 q 的平面上的轴转动。

$$[0 \ 0 \ 0 \ q_x \ q_y \ q_z]v_1 = 0 \quad (3.4.6)$$

约束器五：POR 约束在一条线上

为了避免 POR 同时在两个方向运动, 我们可以使用两个约束器拷贝, 每行一个 (见图 3.4.1 #5)。

$$\begin{bmatrix} a_x & a_y & a_z & 0 & 0 & 0 \\ b_x & b_y & b_z & 0 & 0 & 0 \end{bmatrix} v_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad (3.4.7)$$

约束器使 POR 速度在 \mathbf{a} 和 \mathbf{b} 方向上都没有速度分量。这样的结果就是 POR 被约束为只在 $\mathbf{a} \times \mathbf{b}$ 这条线上移动。

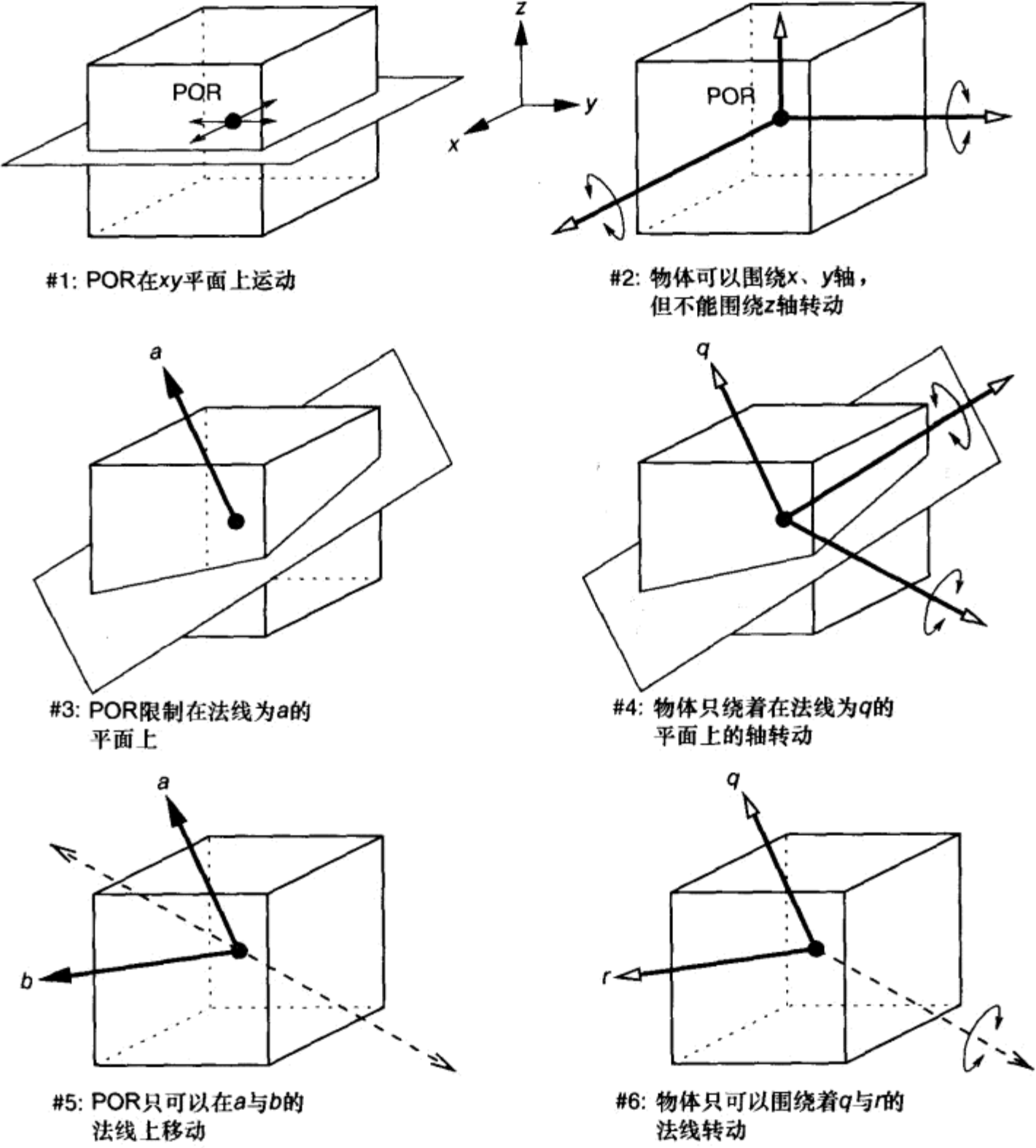


图 3.4.1 约束器构造模块

约束器六：围绕一条固定轴的旋转

与约束器五相似，我们可以阻止同时围绕两条轴转动，把转动约束到一条轴 $\vec{o} = \vec{q} \times \vec{r}$ （见图 3.4.1 #6）。

$$\begin{bmatrix} a_x & a_y & a_z & 0 & 0 & 0 \\ b_x & b_y & b_z & 0 & 0 & 0 \end{bmatrix} \vec{v}_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{3.4.8}$$

约束器七：固定的 POR 或旋转

如果我们把约束器五延伸为 3 行，可以得出下面这条约束器方程式：

$$\begin{bmatrix} a_x & a_y & a_z & 0 & 0 & 0 \\ b_x & b_y & b_z & 0 & 0 & 0 \\ d_x & d_y & d_z & 0 & 0 & 0 \end{bmatrix} \mathbf{v}_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \text{ 或 } \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix} \mathbf{v}_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad (3.4.9)$$

这是一个明显的状态： $v_{ix} = v_{iy} = v_{iz} = 0$ 。运用这个约束器的结果就是物体的 POR 保持固定在空间中一个位置。如果这三个向量不是互相独立的（例如一个向量是另一个的大小倍数关系），那么模拟可能变得不稳定，因为模拟将无法决定沿每个约束器方向要施加多大的力。我们可以应用一条类似的法则来约束所有的旋转运动。值得注意的还有，我们可以结合约束器七和约束器六来创建一个简单的单个物体铰链。

3.4.3 创建有用的游戏约束器

前面介绍过的例子展示了约束器的基本原理。现在我们要讨论如何把这些思想进行实际运用，建造可能运用于游戏的约束器。在这个过程中，我们会看到如何建造二体约束器，如何运用向量 \mathbf{c} ，以及如何运用限制力来制造硬性碰撞、发动机及刹车。

1. 滑轮和绳子组合的约束器

假设一个游戏场景：有两个平台由一根绳子和两个滑轮相连（见图 3.4.2）。向量 \mathbf{a} 和 \mathbf{b} 指向平台可以移动的方向。如果游戏者站在平台 A 上，由于重量的增加，平台会下降，而平台 B 则会上升。

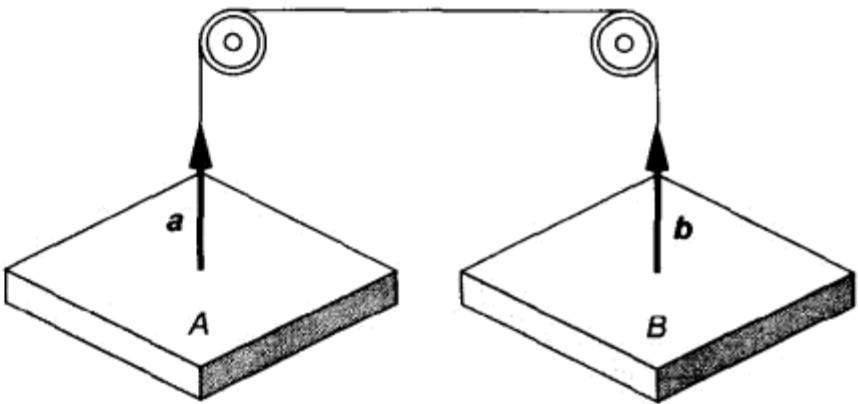


图 3.4.2 滑轮和绳子组合的约束器

为了增加真实性，下降的速率应该与游戏者的重量成比例，并且任何额外加在 B 上的重量都应当使下降速度减慢。这样的效果可以通过运用简单的游戏逻辑创建，但是不幸的是，每一个额外的真实物理现象通常都需要由一个额外的代码来控制。但是，假如使用了刚体模拟，所有这些物理行为（甚至更多的东西）都不会产生额外的花费。

那我们如何创建一个约束器来为这种情况建模呢？回想一下，两个物体的约束器有 $\mathbf{J}_1 \mathbf{v}_1 + \mathbf{J}_2 \mathbf{v}_2 = \mathbf{c}$ 的形式。如果设定 $\mathbf{c} = 0$ ，我们就可以说 $\mathbf{J}_1 \mathbf{v}_1 = -\mathbf{J}_2 \mathbf{v}_2$ ，它反映了两个物体沿各自特定方向的速度是相互制约的。因此，滑轮和绳子组合的约束器就成为

$$[a_x \ a_y \ a_z \ 0 \ 0 \ 0] \mathbf{v}_1 + [b_x \ b_y \ b_z \ 0 \ 0 \ 0] \mathbf{v}_2 = 0$$

它表示了物体 1 沿 \mathbf{a} 方向的 POR 速度必须与物体 2 沿 $-\mathbf{b}$ 方向（允许 \mathbf{a} 和 \mathbf{b} 的相对长度）的 POR 速度相吻合。相嵌的齿轮也可以用类似的方法模拟，只是使用角速度的约束器而非线性速度的约束器。

2. 螺旋约束器

螺旋约束器能迫使物体在沿一特定方向运动的时候发生旋转，比如螺帽沿着螺母的纹路

运动。这种效果可以通过使用一行包括线性和角度的约束器来实现，这样线性运动和角度运动就可以相互制约了。例如，约束器

$$[a_x \ a_y \ a_z \ q_x \ q_y \ q_z] \ v_1 = a \cdot \dot{p}_1 + q \cdot \omega_1 = 0 \quad (3.4.10)$$

被分解为线性部分和角度部分来反映沿 a 轴的线性速度必须是沿 q 轴角速度的倍数。在螺旋约束器中， a 和 q 指向同一方向。

3. 球窝关节

球窝关节有很多用途，比如连接成一条链子，或者连接布娃娃的手脚。图 3.4.3 展示了连接器，其中 g_1 和 g_2 是从物体 1 和物体 2 的 POR 到球体的向量。从 u_1 、 u_2 、 u_3 各方向看，我们希望球体从物体 1 和物体 2 测得的速度是一致的，因此在每个方向的向量上我们得出 $u_i \cdot (\dot{p}_1 + \dot{g}_1) = u_i \cdot (\dot{p}_2 + \dot{g}_2)$ 。由于这个约束器是相对于球形而言的，所以我们必须用漂移法则 (shifting rule) 来使它与物体的 POR 相关。

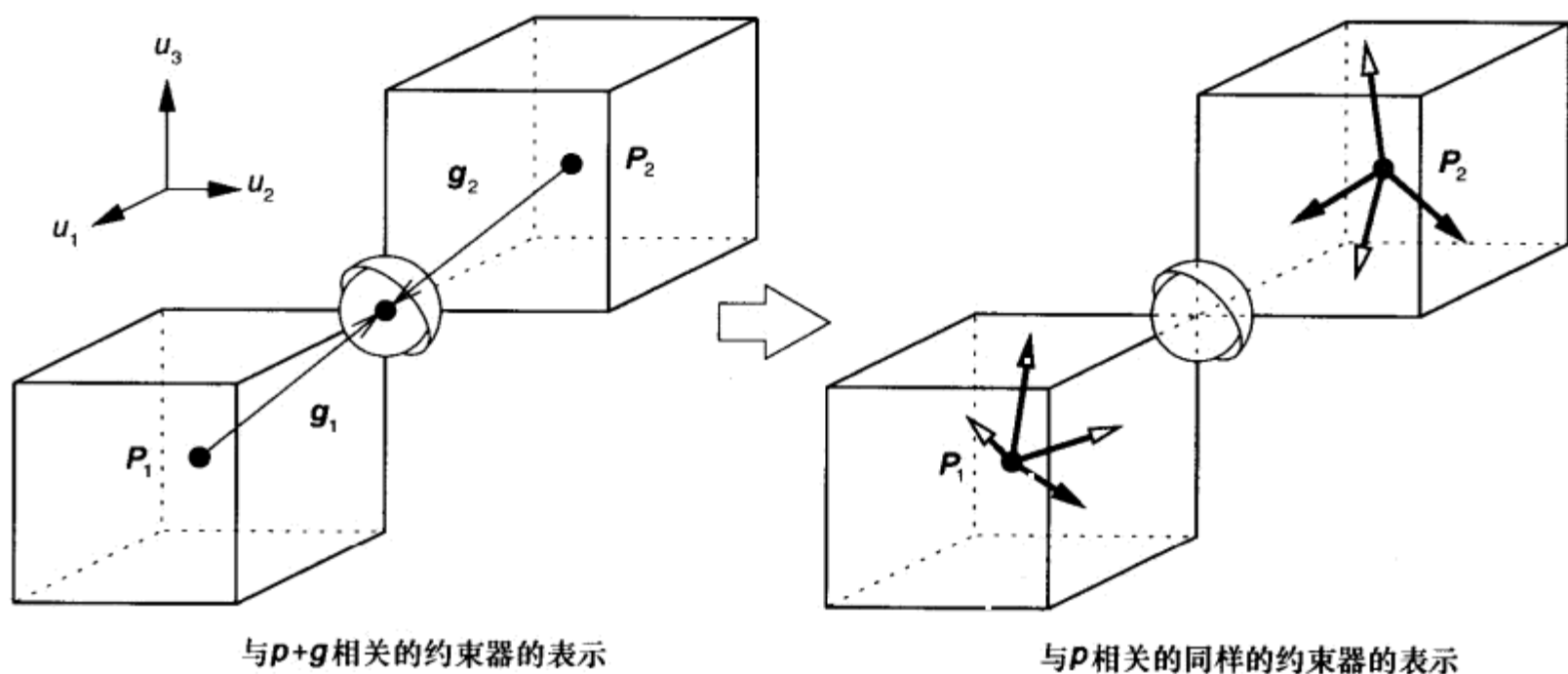


图 3.4.3 用漂移法则创建的球窝关节

根据漂移法则，如果约束器是专门针对 g_1 点和 g_2 点并和物体 1 和物体 2 相关，如以下式子所示：

$$a_1 \cdot (\dot{p}_1 + \dot{g}_1) + q_1 \cdot \omega_1 + a_2 \cdot (\dot{p}_2 + \dot{g}_2) + q_2 \cdot \omega_2 = c \quad (3.4.11)$$

那么与其等价的相对于 POR 的约束器就是

$$a_1 \cdot \dot{p}_1 + (q_1 + g_1 \times a_1) \cdot \omega_1 + a_2 \cdot \dot{p}_2 + (q_2 + g_2 \times a_2) \cdot \omega_2 = c \quad (3.4.12)$$

漂移法则准确地使在 POR 的线性速度和角速度相互约束。把它应用在每个 u_i 的球窝关节，有

$$u_i \cdot \dot{p}_1 + (g_1 \times u_i) \cdot \omega_1 - u_i \cdot \dot{p}_2 - (g_2 \times u_i) \cdot \omega_2 = 0 \quad (3.4.13)$$

我们只要用同一方法计算 $u_1 \dots u_3$ ，把全部 3 行约束器加在一起，就得出了球窝关节。

$$\begin{bmatrix} 1 & 0 & 0 & 0 & g_{1z} & -g_{1y} \\ 0 & 1 & 0 & -g_{1z} & 0 & g_{1x} \\ 0 & 0 & 1 & g_{1y} & -g_{1x} & 0 \end{bmatrix} \mathbf{v}_1 + \begin{bmatrix} -1 & 0 & 0 & 0 & -g_{2z} & g_{2y} \\ 0 & -1 & 0 & g_{2z} & 0 & -g_{2x} \\ 0 & 0 & -1 & -g_{2y} & g_{2x} & 0 \end{bmatrix} \mathbf{v}_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

4. 铰链连结器

铰链连结器其实就是多了两行约束器的球窝关节。多出的两行约束器迫使物体沿铰链轴排列起来。它们是对约束器六的概括，以避免物体沿轴和轴相对互相旋转。

$$\begin{bmatrix} 0 & 0 & 0 & q_x & q_y & q_z \\ 0 & 0 & 0 & r_x & r_y & r_z \end{bmatrix} \mathbf{v}_1 + \begin{bmatrix} 0 & 0 & 0 & -q_x & -q_y & -q_z \\ 0 & 0 & 0 & -r_x & -r_y & -r_z \end{bmatrix} \mathbf{v}_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad (3.4.14)$$

5. 一堆物体的硬接触

通常情况下，游戏中会出现一堆物体使游戏者可以推动、穿过或者进行爆破。要把刚体堆砌起来，我们需要对它们之间的非穿透接触（nonpenetration contact）建模。惩罚性方法把由使用者计算的抑制力应用在各个物体上，它调试起来困难，而且可能引起不稳定。而真正对我们有用的是具有以下特性的硬接触约束器。

当两个物体被推向碰撞点的时候，约束器以同等大小的力把它们推回去；而当物体被推开的时候，约束器不做反应。这个可以通过约束力限制区间（constraint force limit）达到。

首先，我们先从总体上看一下力的限制区间（force limit）是怎样起作用的。应用在物体1和物体2上的约束力（constraint force）向量 \mathbf{f}_{c1} 和 \mathbf{f}_{c2} 是与物体相关的约束器对列的加权总和。在 m 排约束器队列中，有权数 $\lambda_1 \dots \lambda_m$ （由模拟计算出来的拉格朗日乘法器的值）以及

$$\mathbf{f}_{c1} = \mathbf{J}_1^T \boldsymbol{\lambda} \quad \mathbf{f}_{c2} = \mathbf{J}_2^T \boldsymbol{\lambda} \quad \text{其中, } \boldsymbol{\lambda} = [\lambda_1 \dots \lambda_m] \quad (3.4.15)$$

在一般无限制区间的约束器中，解析器（solver）允许 $\lambda_1 \dots \lambda_m$ 取任意值。也就是说，解析器会“拿到什么解析什么”以保证满足约束器。力的限制区间给解析器以约束，使得它只能在一特定的区间内应用约束力，例如， $\lambda_{iLO} \leq \lambda_i \leq \lambda_{iHI}$ 。如需要用到更大的力，那么 λ_i 就会取力的上限值，因而约束器的需要就不能完全达到。如果应用了限制区间，模拟器必须在每一个时长内解决线性补数问题（LCP）[Murty88]。像 ODE 这样的模拟器允许在任何约束器队列上实施限制区间。

要建造一个接触约束器，我们可以把 POR 漂移法则应用在约束器三的两个物体的版本中，这样物体在接触点就会受到一个力 $\lambda \mathbf{a}$ 。我们说如果 λ 是正的，那么 $\lambda \mathbf{a}$ 就会把两个物体推开。如果我们设 $\lambda_{LO} = 0$ ， $\lambda_{HI} = \infty$ ，并且所需的 λ 是正的，那么就会有一个正确的反作用力使两个物体分开。如果所需的 λ 是负的，这个作用力就会变成零，使两个物体不受阻力，可以分开。通常情况下，为了使两个物体保持分开，会需要多个接触点；但是这取决于物体的几何特征。

这里介绍一个很巧妙的小窍门，如何为有“磁性”的物体建模：设 $\lambda_{LO} = -M$ ， $\lambda_{HI} = \infty$ 。它的意思是，施加非零作用力 M 把物体拉开。

6. 发动机建模

要让车辆行驶需要某种发动机。给发动机建模最简单的方法就是在轮子和底盘之间施加

一个力矩。这样虽然行得通，但是也有问题——特别是，当使用大的力矩时，可能会引起不稳定。另外一个模型使用了一个约束器队列，它使用了一个不为零的值 c ，在轮子和底盘间设了一个匀速角速度。总的说来，当 c 不为零的时候，它把物体的速度设为沿着被约束的方向（例如，在约束器三中设 $c = c_z$ ，就是说沿 a 的 POR 速度一定是 c_z ）。我们来看看把约束器六和约束器七结合起来得到的这个单个物体连接器，另外再增加下面这一列：

$$[0 \ 0 \ 0 \ s_x \ s_y \ s_z]v_1 = [c_s]$$

其中 $s = (q \times r) / |q \times r|$ 是铰链旋转轴的单位长度。这是一个“强有力”的铰链，它围绕铰链旋转轴有角速度 c_s 。但是，也正是因为它无穷强大，使它成为一种可怕的发动机模型，因为无论是怎样的约束力矩也阻止不了它达到目标速度。

只要在发动机约束器队列中设 $\lambda_{LO} = -f_{\max}$ ， $\lambda_{HI} = f_{\max}$ ，那么，这个问题还是可以解决的。发动机不能使用超出 $\pm f_{\max}$ 的力矩来达到目标速度。如果发动机能在一个时长内使物体的速度达到理想水平，这样最好，发动机会这样运转；否则，它会运用最大可能的力矩使物体在几个时长使物体的速度达到要求。这样，任何外加的力矩都可以与约束器力矩对抗，使发动机受的阻碍就像在真实情况中的一样。把 f_{\max} 设为零使发动机失效，有效地解除了发动机约束器队列的作用。在 f_{\max} 较大的情况下，这个发动机和直接作用力矩发动机（direct-torque motor）作用相当，却稳定许多。此外，它也是较合理的附加变速箱的引擎模型，因为内部摩擦力限制着最大速度（如典型的汽车动力传动系统）。

7. 刹车系统及静摩擦

车辆不但需要发动机，也需要刹车系统。和发动机类似，最简单的解决刹车的方法就是直接作用在轮子上的力矩。同样，这也会导致不稳定。车轮碟刹上的制动力矩是由静摩擦这样一个持续不变的抵抗角速度的力矩产生的。通过用一个目标速度被设为零的有力限制区间的发动机，静摩擦力就可以很好地被建模。这个约束器会使用一个不超过 $\pm f_{\max}$ 的力矩使连接器的速度为零。这里， f_{\max} 是根据刹车脚踏被压下的程度设定的——当不需要任何制刹的时候， f_{\max} 为零，约束器不起作用。如果你已经有了一个有力限制区间的发动机来驱动车轮的话，那么就等于白捡一个刹车系统，因为它们可以使用同一个约束器。


8. 修正位置性误差

假设模拟器十分完美的话，我们只需把所有物体置于各自的起始位置，加上一些约束器，那么约束器就会时时刻刻都准确无误地工作了。但在实际操作中，模拟器并不是完美无缺的，数值误差会影响物体的位置和运动方向，尤其是当旋转速度很高的时候。防止这些发生的一个有效方法就是允许约束器本身在物体运行不正常的时候把物体调整到正确的设置上。这可以通过在 c 中使用非零值来实现。

约束器不能直接影响一个物体的位置，它只能改变其速度。然而，如果一个约束器的“目标”是为了保持一个恒定的位置（或者是为了使两个物体的相对位置保持恒定），那么 c 的设置应当使约束器能够迫使物体从不正确的位置回到理想位置。例如，在约束器一中，如果目标是保持 $p_{1z} = 0$ ，那么我们可以设 $c_z = -kp_{1z}$ ，这样更正速度和位置误差是成比例的。这能随着时间的过去，使位置误差以一个由 k 决定的指数数率减少。要注意，如果 k 太大了，更正

后的位置会在一个时长内超出合理的范围,这可能会引起不稳定。总的说来,我们在每个约束器队列中设 $c = -\text{error}/h$, 这里误差的测量是沿着方向矢量或者相应队列 J 所对应的轴。例如在球窝关节中,误差的测量是沿 $u_1 \dots u_3$ 方向的。

3.4.4 光盘中的内容

 随书附送的光盘中包含了文章中提到的所有约束器的例程源代码,它们使用 ODE。光盘还包含了完整源代码、文件,以及为 ODE 所做的例程程序。如果读者有兴趣创建自己的约束器,可以参阅源文件,里面有一些本文章没有涉及到但很实用的细节。

3.4.5 结论

这篇文章阐述了怎样使用各种简单的约束器行来建立新的刚体速度约束器。利用这些构造模块,加上漂移法则、对 c 向量的理解,以及力限制区间,读者几乎可以建造任何约束器。同时,使用由使用者设计的约束器和标准约束器的模拟库,如 ODE,能让使用者具有为新情况建模的强大能力。

3.4.6 参考文献

[Baraff96] Baraff, David, "Linear-time Dynamics Using Lagrange Multipliers," *Computer Graphics Proceedings, Annual Conference Series*: pp. 137–146, 1996.

[Hecker96] Hecker, Chris, "Rigid Body Dynamics," available online at www.d6.com/users/checker/dynamics.htm.

[McMillan94] McMillan, Scott, *Computational Dynamics for Robotic Systems on Land and Under Water*, Ph.D. Thesis, The Ohio State University, Columbus, OH, 1994.

[Murty88] Murty, Katta G., "Linear Complementarity, Linear and Nonlinear Programming," available online at http://ioe.engin.umich.edu/people/fac/books/murty/linear_complementarity_webbook/, 1988.

[Smith03] Smith, Russell, "The Open Dynamics Engine," available online at <http://q12.org/ode/>.

[Witkin97] Witkin, Andrew, and David Baraff, *Physically Based Modeling: Principles and Practice* (Online SIGGRAPH '97 Course notes), available online at www-2.cs.cmu.edu/~baraff/sigcourse/index.html.



3.5 在动力学模拟中的快速接触消除法

作者: Ádám Moravánszky 和 Pierre Terdiman, NovodeX AG

E-mail: adam.moravanszky@novodex.com, pierre.terdiman@novodex.com

译者: 李鸣渤

审校: 许竹钧

典型的物理管线包含三个特有的部分。

- **碰撞侦测器**: 侦测一个场景中物体之间的碰撞。
- **接触发生器 (contact generation)**: 通过碰撞数据创建接触点。
- **动力模拟**: 增强用接触来表现的非穿透性约束器, 并更新物体的状态。

要模拟发生接触中的物体的互动, 非穿透性的约束器必须以紧凑的方法表达, 这样才能对由此得出的动力进行有效的模拟。为达到这一目的, 自从 [Lozano-Perez83] 之后有关机器人技术的著作中就引入并使用了接触点的概念; [Hahn88] 及其他一些有关电脑图形的著作是早期介绍及使用这一概念的地方。接触点反映了碰撞体之间的本地非穿透性约束器。使用的接触点的数目对准确性、稳定性以及模拟的速度都有很大的直接影响。给每一对碰撞物体只创建一个接触点通常情况下是不够的; 反过来说, 根据管线中动力部分的运用情况, 过多的接触点则要么导致稳定性方面的问题, 要么造成性能的降低。这篇文章就是要阐述几种减少接触的算法, 来减小接触点的使用个数及与之相关的工作量, 如: 接触预处理 (contact preprocessing)、接触分类集中 (contact clustering), 以及接触的保持 (contact persistence)。

3.5.1 减少接触

碰撞侦测算法通常把网格 (mesh) 分解成多个三角形或者体积元素 (volume element), 然后单独为每一个分解出来的元素建立接触点。这样, 当有无数相互交叠的元素时, 系统就会因为太多的接触点而崩溃, 特别是当这些元素的分辨率很高, 或是物体之间相互穿透很厉害的时候。

即使侦测到的接触点充分地显示了在碰撞中本地几何体上特定位置, 要创建为数众多的接触点还是有它的不足之处。首先, 由于众多的接触产生超额的计算量, 用以解决真正接触力的动力算法在计算起来时比较困难。这些计算量至少是接触点数量的 4 倍 [Baraff92]、[Anitescu99]。

在这种情况下，每增加一个不是完全必要的接触点都是非常不明智的。此外，矩阵要表示这样一个冗重的系统必然出现问题。解析器（solver）在对矩阵进行取因数操作的时候也会出现困难。有些迭代解析器（iterative solver）用以解决收敛的迭代可能比相同数量的线性独立的约束器用的迭代还多。

实际上这已经不是什么困难了，因为用于模拟互动性动力学的近似算法线性地减少了约束器的数量 [Jakobsen01]。但是，这些成本不高的算法经常对接触点的数量比较敏感。因为它们运用的方法很典型，就是对每个接触施加惩罚力或者冲量，而当接触点数量出现差异时，就会出现一些奇怪的现象。基于惩罚方法的解析器在游戏中仍然广泛使用。其中一些方法，对单个接触案例，使用了各种常量和可微调的门限值，例如，在平面上滚动一个球体。它们在处理这样的理想案例时能正常运作，然而，当同一时间发生的接触数目增加的时候，模拟将会与理想的案例的差别越来越大，模拟会变得越来越不稳定。以一个放在平面上的圆环为例，如果在每个顶点都建立一个接触，其产生的力和理想状态就相差甚远，直接取决于网格。通过减少接触的数量，有助于解决稳定性的问题 [Kim02]。正是因为这些原因，接触点的数量应该保持最少，以确保模拟的真实性。我们可以分两步来解决这个问题，即通过预处理碰撞形状以及对接触点进行分类集中。

1. 预处理

接触的种类至少有以下两种：

- 一种是可以不去处理的接触，因为这些接触对整个模拟没有影响；
- 另一种可以合并为单个接触，这样的模拟准确性可能有所降低，但仍然可行。

接触的预处理解决第一种情况。预处理步骤检查一个已知形状的所有可能接触。但也不总是这样的，要根据具体的物体的形状而定。然而，如果是的话，在模拟中不产生显著作用的接触会被标示出来，并且事先被弃置。这个方法十分有用，因为除去多余接触的最佳方法就是从一开始就不产生它们。

2. 分类集中

接触的分类集中用来解决以上提到的第二种接触。通过减少类似接触点，降低了接触点的数量，减少后非穿透性约束器的特性不会改变。在最简单的情况下，例如靠在平面上的一个物体，所有有效的接触法线都是相同的，并等于平面的法线；所有接触点都位于平面上或者在平面下方。接触低于平面的距离就是在那一点上的穿透深度。在不考虑穿透的情况下，为了达到硬体模拟的效果，我们可以得出，只要接触时与平面产生的凸面形状是相同的，接触的情况就是相同的。

实际上，靠在平面上的物体只可能做三种动作。

- 它可以从平面上离开；在这种情况下，接触表面会突然消失。
- 它可以向平面挤压，最终沿着平面滑行。
- 它可以沿着平面绕着凸面体的边滚动落下。

因此，影响碰撞行为动作的只有普通碰撞法线方向，以及碰撞点产生的凸面的边。所有不在凸面体上的碰撞点都可以忽略不计，而这是不会改变非穿透性约束器的性质的。

通常，能进一步减少凸面内的接触点是最好的，特别是在像图 3.5.1 所示，圆柱体和地

面之间发生碰撞的情况下。我们已经研究出一种算法来去除最少的凸面的顶点，这样做是因为它反过来会把圆柱体不该下落时发生下落的可能性减至最低。可是，一成不变地实现这一严格的方法代价太昂贵；实际上我们会对这个算法加以改良，通过由坐标轴圈出碰撞点的范围约算出凸面体。这样的粗略计算无论是用在单个的物体上（见图 3.5.1），还是复合的物体上（见图 3.5.4），工作效果都是相当不错的。

3. 持续性

在减少了接触的数量后，算法会使之保持持续状态。由模拟的每一帧得出的接触点都有很强的短暂连贯性；也就是说，由一段连续的帧画面得出的接触点都十分类似。清楚地观察一个时间段内的类似状况是很有帮助的。首先，一些解决碰撞力量的迭代算法可以硬性地从 一个初始估算的力开始：估算值越接近，它就越趋近于实际状况。这个过程我们称之为热启动。我们在前一个时长内热启动具有接触力的解析器，希望当前帧的接触力和前一帧的接触力相似。但是，这只有在之前的接触点和当前接触点间存在联系的情况下才是可行的。

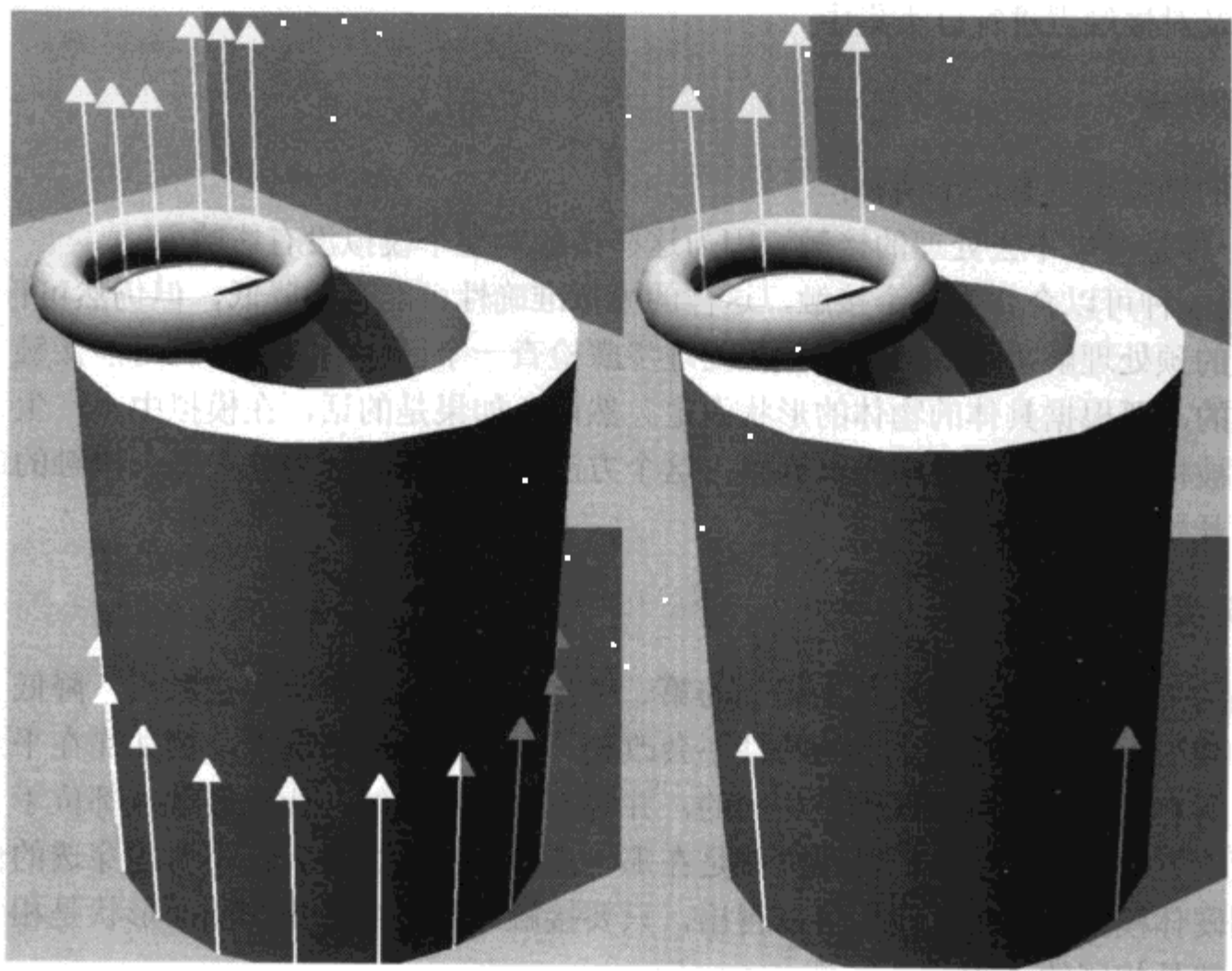


图 3.5.1 消减之前和之后的接触集合，
圆柱体和圆环面是作为靠在平面上的网格来处理的

对持续性接触的另一应用是对根本不做相对滑动的两个物体间的静摩擦力尽可能地做出最佳模拟。要模拟出这种条件，我们可以通过保证，在每个时长中物体碰撞点之间的相对切线速度为零而得到。当发生的错误累积起来的时候，物体就会沿着接触的平面切线慢慢地滑行。在我们用基于拉格朗日乘法器的技术来模拟物体间连结器的时候，也会出现同样的

情况：空间位置的错误会在连接器积累起来。要消除这个错误，我们必须能对它进行测量。在连接器中，要计算这个错误很容易，因为我们知道连接器应该在哪里与每个物体的本地坐标系相接。在接触中，除非我们知道接触在创建的时候位置在哪儿，或者静力摩擦的条件什么时候开始作用，否则这是不可能做到的。持续性接触就是解决这个问题的一种方法。

3.5.2 对预处理的详细分析

减少接触点的最佳方法就是从一开始就不去生成这些接触点。而具体把这个理念实施起来的时候就要看具体所使用的碰撞检测算法的类型了，因此，用一个例子来表示：四方体接触点生成算法，用它来处理游戏物体与静态环境物体的碰撞。

大家都希望在检测碰撞时使用的数据与用在图形中的数据相同，并且其中一种最常见的图形显示格式是一个索引过的三角形列表。这些三角形被认为是有方向的（它们的其中一面被当作是正面——也就是让外界看见的一面，另一面则是背面），但是这必须是它们满足的惟一条件。如果网状体包含有许多非多面共有的边或是 T 形交叉的话，这种方法的效果会减弱。

假设碰撞检测系统能报告所有与四方体相交的网状体的三角形的话，碰撞的生成就只受四方体的形状以及每个三角形的影响了。这包含了对每个三角形边的测试以及形状中顶点的测试。沿着两个与碰撞形状都相交的三角形所共有的边，这种方法会产生重复的接触。

以下几种网状体的边根本不值得测试，可以在预处理的时候把它们标示为非活动的状态。

- 内部边（interior edge）就是两个三角形间有着相同法线的边。这样一条边只能用来把平面多边形分解成三角形网格；它并非一个实际存在的几何特性，因此可放心地忽略。

- 凹面边（concave edge）就是两个三角形之间与三角形的正面构成凹角的边。这条边在一般的状态下不能被另一个物体触及，除非它也接触到了两个三角形中的一个，因此也不需要对其进行测试。

这样一来，就剩下两种碰撞检测系统一定要测试的边了，边界边（boundary edge）（不被两个三角形共有的边）和凸面边（既不在内部也不凹入的非边界边）（见图 3.5.2）。测试凸面边时，只从包含它们的其中一个三角形出发就足够了。非多面共有的边的测试和凸面边是一样的，因为我们假设，出现这种情况的频率不高，不需要引起特别关注。

一些顶点也可以用类似的方式忽略不计。

- 把所有与凹面边相连的顶点当成凹面，不去管它们。这与忽略凹面边的原理是一样的。

- 把所有只与内部边相连的顶点当成内部点，忽略不计。

- 对与之相连的其中一个潜在三角形的其他所有顶点进行测试。

利用这些法则可以免掉碰撞检测计算中很大数量的特性，因此也就在减少所产生的接触点数量的同时缩短了计算时间。

这些法则在预处理的步骤中实现是最佳的。首先，给每个三角形分配 6 位的额外存储空间。三角形的每一个顶点及边分得 1 位，且所有顶点及边都只有在设定了相应位之后才能参

与到接触计算当中。开始时，三角形的所有 6 位都被清除。网状体进行详细演算，并为三角形的每一条边做记录。每条边的记录中包含两个顶点的引用和一个三角形的引用。然后这些边会被分类处理，这样有着相同的两个顶点的两条或多条边在列表中是相连的。对分类后的列表进行详细研究。当系统运行相同的两个顶点之间的边的时候，只对其中一条边进行测试：如果是凸面边或者边界边，在边的三角形标记记录中设置 3 位与边和它的两个顶点相对应。如果边是凹面的，在边的排序列表中把它的记录标示出来，但是不进行其他处理。当处理完边的列表中的所有边之后，进行第二轮审查：在这轮审查中，找到作了标识的凹面边，并在它们出现的所有三角形中清除其顶点标记。这样的程序能确保所有与一条凹面边相邻的顶点都不被检测。

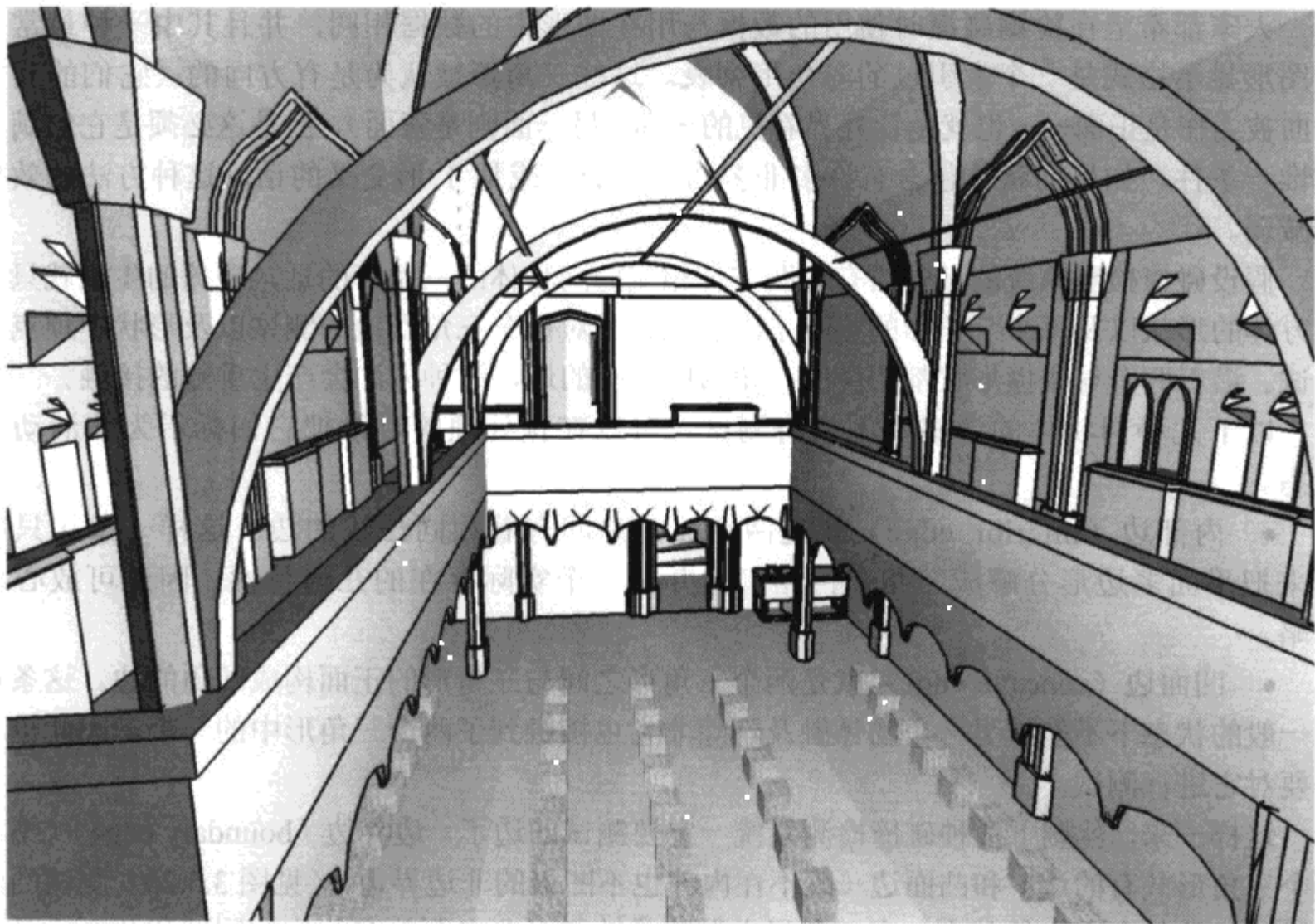


图 3.5.2 只有粗黑的边才需要进行碰撞检测（感谢 Arkane Studios 提供游戏场景）

3.5.3 对接触的分组群的详细分析

我们的接触分类集中算法的第一步就是根据接触法线把接触点分组。有些类型的碰撞检测能生成相同法线方向的全部接触点。有两个例子，一个是任意类型的凸面形状间的碰撞（见图 3.5.3），另一个是一个平面和任意形状间的碰撞（见图 3.5.1）。在这种情况下，法线分组生成的步骤是微不足道的，可以跳过。否则，至少可以使用两用方法：立方体映射算法以及基于 k-means 的算法。第一个适用于包含一些法线方向完全不同的一组接触；第二个更合适于没有两个几乎等同的法线的平滑形状。

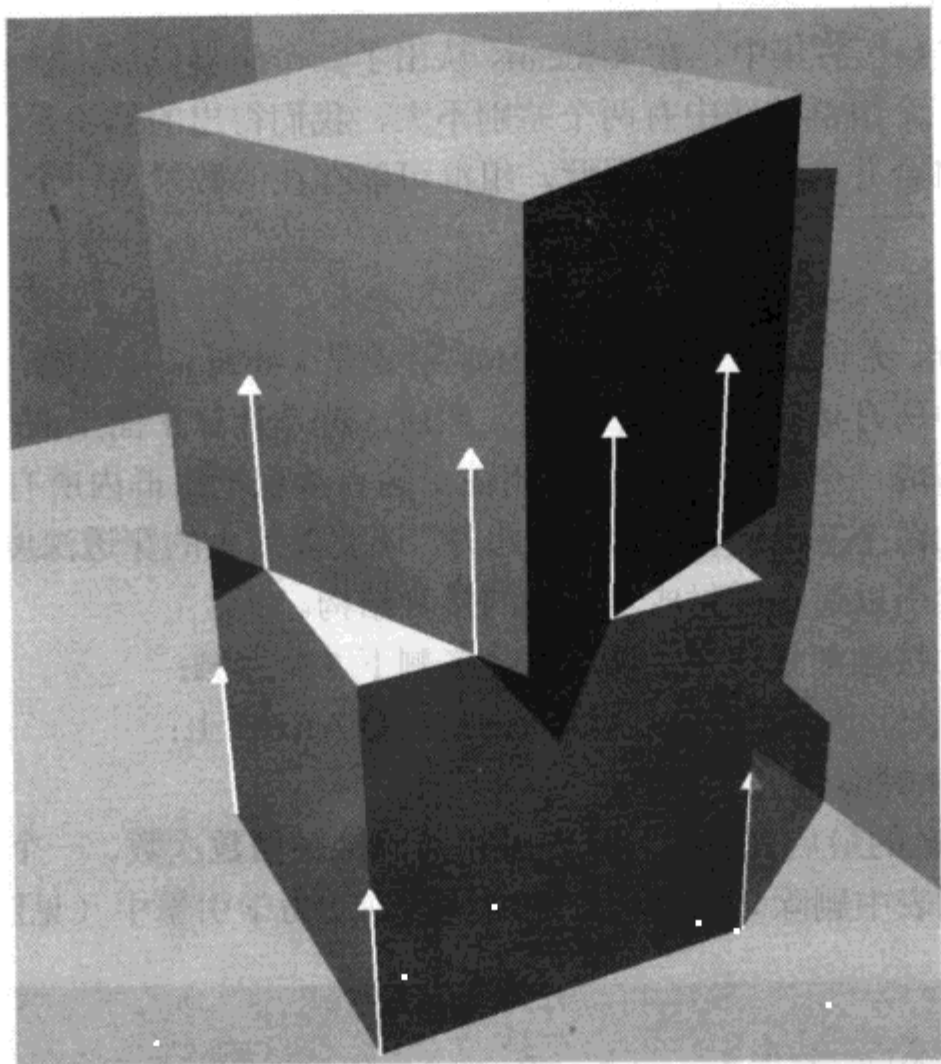


图 3.5.3 两个凸形物体在碰撞中产生接触面的法线方向相似

1. 立方体映射的分组群

这种算法由一组给出的输入接触及一组离散的接触群开始。每一个群代表有着相似法线的一组接触。每个对应群的法线可以从立方体映射表中查找。这种查找通常要比普通的法线向量量子化快。单位立方体的 6 个面进一步被分为等边的 $N \times N$ 格子，形成 $6N^2$ 个群。这种映射和用于法线掩码 (normal mask) 的类似 [Zhang-Hoff97]。接触群不会被明确地分配，这样可以节约存储空间。我们的做法是，在空闲的时候计算群的索引，然后用来给记录过的接触分类成群。然后，我们再按顺序处理每一个群，边处理边减少。

2. K-Means 的分组群

基于立方体映射的算法在法线很集中的时候能很好地运作。否则，一个组群的法线会从群内分离开来，落在不同的立方体映射单元内。这种情况在进行圆形物体，如球体，以及被分为许多小立方体的几何体的碰撞检测时可能发生。这种情况也会产生许多接触点，并且是很理想的简化选择。k-means 分组群算法是很有名的数据分析工具。给定一输入向量的集合，以及要找出的群的数量 k ，它就能快速地把输入向量映射到 k 组群上。同时，组群的中心也确定出来了。在我们的应用中，接触法线就是输入向量。要事先知道法线如果能形成“自然”组群，形成了多少个是很难的。在一个球体对一个具有高分辨率模型的情况下，有三个组群在实际中表现良好。要求太多数量的组群会导致自然组群被分离，并增加算法的运行时间。而只有一个组群则相当于简单地取所有法线的平均值，会导致反向的法线被抵消。

根据我们的经验来看，如果在自然组群给出输入向量，通过硬性设置初始化组群的方法，k-means 就会在三次叠代后集中。在 k-means 认出了三个组群以后，检查每一组中产生出来的平均法线。如果三个组群法线中有两个差别不大，我们得出的结论是，自然组群的数量低于三个，同时我们将合并这些组群。因此，组群可能存在的数量是一个、两个，或者三个。

3. 组群的减少

对于每一个组群，无论组群是通过立方体映射还是 k-means 找出的，都要计算投影平面，把它当作这个组群中所有接触平面的平均值。然后，就是计算平面上两个标准正交的向量，与平面的法线向量组成一个基准。接下来，用简单的点乘积把组群内所有接触投影在平面上，并记录沿以上得出的两个基准向量的极值。此外，还要以最大的穿透深度记录接触。实际上，读者可以使用接触索引以避免应付成本更高的接触结构。

当组群里的全部接触都处理完了以后，最多剩下 5 个接触：

- 其中 4 个接触位于平面上接触投影周围的 2D AABB 上；
- 还有 1 个接触拥有最大的穿透深度。

有时候，5 个接触的最后列表实际包含了相同的接触倍数次数。一个小的 $O(n^2)$ 循环最终会把多余的接触从列表中删除。剩下的接触则被送到动力学引擎中（见图 3.5.4）。

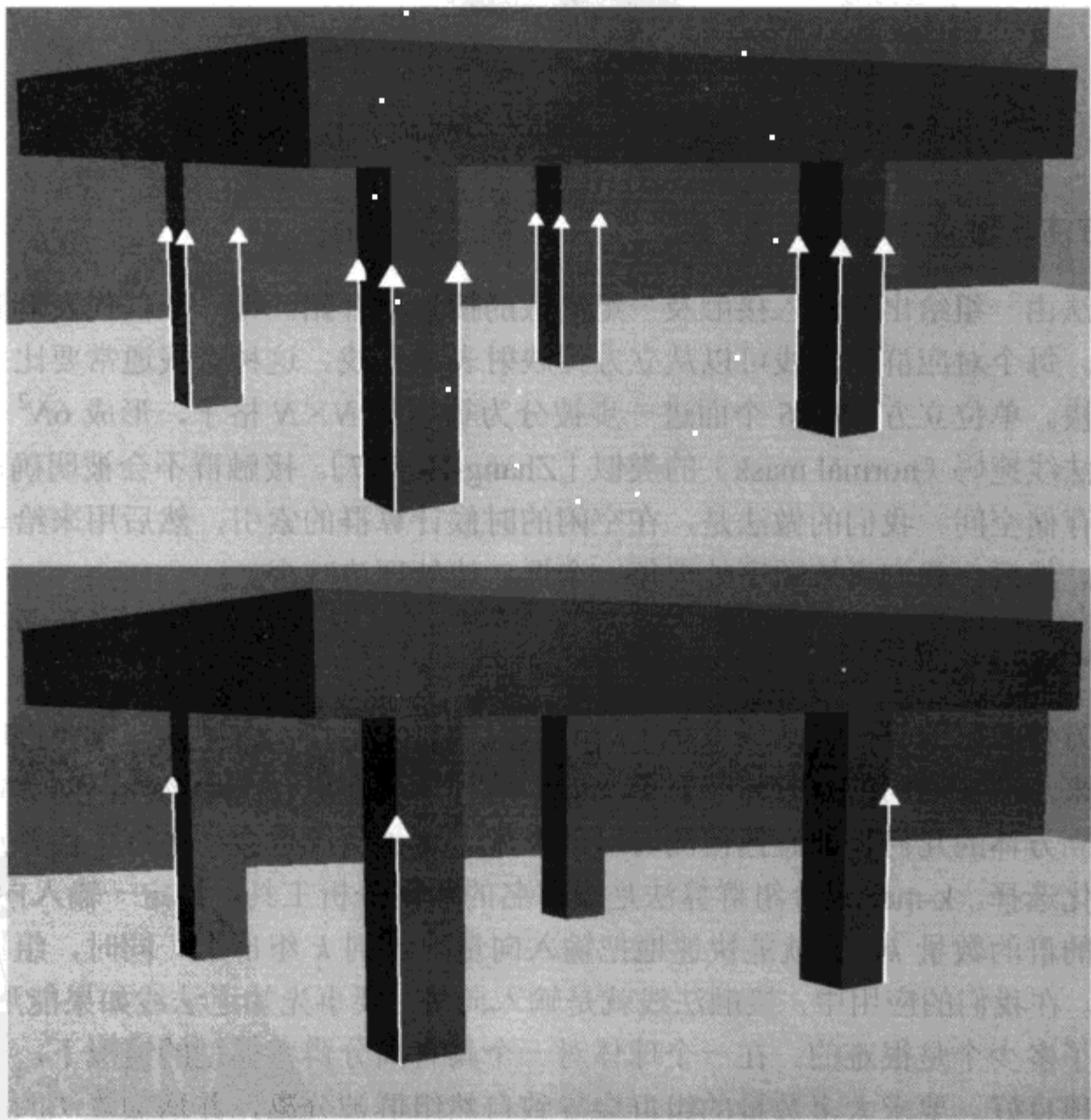


图 3.5.4 组合物体也可以从消除接触中获益

3.5.4 对持续性的详细分析

如果一个物理引擎还未支持持续性的接触，那么可以通过散列法来实现。在一个模拟时长中，接触被记录到一个双缓冲的接触向量中。双缓冲保证了从前一个时长中得到的接触在当前时长中还是可用的。然后，两个接触列表被送到处理持续接触的代码中，以找出它们的一致性。

1. 接触标识符

接触结构原本包含了发生接触的两个物体（要么用句柄要么用实际的指针）。要实现持续接触，要把一个 64 位的接触的 ID 加入到结构中。生成接触例程（也就是负责填写接触结构的一部分代码）此时也负责创建一个惟一的标识符来描述每个接触。典型的 64 位 ID 是由两个 32 位的部分组成的，每个部分负责建立接触的物体特性（顶点、边、面）。这些标识符只取决于创建它们的接触生成代码，只要它们定义一个已知的接触的特性并不明确时，可以被任意地设置。其实，特性所指的不仅仅是标准顶点、边，或者面——比如说，voxel 索引也属于其范畴。

2. 散列法

首先，给前一时长的所有接触创建散列值，并把其存储于一散列表中。然后，从当前的接触进行循环，并把其散列放入前一个表格，以搜索一个相近的接触。这种算法的关键是散列值中对目标物体和接触标示符的使用。这种方法相对于另一种给每一对（也就是一对碰撞物体）保存持续性数据（persistent data）的方法有无数的优点。

- 它在与已有的物理引擎结合时所需的修改是最少的。
- 每个接触例行过程的主要散列部分只有一页外加几行的代码用来生成 ID。
- 它的速度很快（实验说明是时间复杂度为 $O(n)$ 的行为）。
- 它很安全，因为它不对位置或者法线进行检查来匹配旧的接触和当前接触。
- 与每个激活对都保持一个持续接触数组的解决方法相对比，它不涉及复杂的内存管理。

而主要的缺点则是，当一个顶点与面的接触变成边与边的接触时，持续性就会消失。

3.5.5 结论

这篇文章介绍了一种在进行近似的凸面计算以前，通过把接触点按法线分组来减少接触点的方法。接触点使用一个基于特性的识别系统来保持持续。在我们自己的工作中，对每一个经过精减后得到的接触点组都要计算一次摩擦力，而不是像我们以前那样对每个接触分别计算摩擦力。这一算法的应用不会对质量有很大的消极影响，反而能有效地在处理复杂的几何模型问题的同时，削减接触点，这反过来提高了性能、增强了稳定性。这种接触消除法的近似特性决定了人们在模拟的准确性上看不出它有所降低，其中主要的原因就是最大穿透的接触点不会被除去。

3.5.6 参考文献

[Anitescu99] Anitescu, Mihai, F.A. Potra, and D. Stewart, "Time-Stepping for Three-Dimensional Rigid-Body Dynamics," *Computer Methods in Applied Mechanics and Engineering*, Vol. 177 pp. 183–197, 1999.

[Baraff92] Baraff, David, "Dynamic Simulation of Non-Penetrating Rigid Bodies," Ph.D. thesis, Department of Computer Science, Cornell University, 1992.

[Hahn88] Hahn, J.K., "Realistic Animation of Rigid Bodies," *Computer Graphics (Proc. SIGGRAPH)*, Vol. 22, pp. 229–308, 1987.

[Jakobsen01] Jakobsen, Thomas, "Advanced Character Physics," Proceedings of the Game Developer's Conference 2001, San Jose, 2001.

[Kim02] Kim, Young J., Miguel A. Otaduy, Ming C. Lin, and Dinesh Manocha, "Six Degree-of-Freedom Haptic Display Using Localized Contact Computations," Tenth Symposium on Haptic Interfaces For Virtual Environment and Teleoperator Systems, March 24–25, 2002.

[Lozano-Perez83] Lozano-Perez, T., "Spatial Planning: A Configuration Space Approach," *IEEE Transaction on Computers*, C-32(2) pp. 108–120, 1983.

[Zhang-Hoff97] Zhang, Hansong, and Kenny Hoff, "Fast Backface Culling Using Normal Masks," *ACM Symposium on Interactive 3D Graphics*, Providence, 1997.



3.6 互动水面

作者: Jerry Tessendorf, Rhythm & Hues Studios

E-mail: jerryt@rhythm.com

译者: 李鸣渤

审校: 刘永静

自1996年以来,用计算机模拟海洋的表面已经作为电影的特色被广泛地使用了,《未来水世界》(*Waterworld*)、《泰坦尼克》(*Titanic*)、《第五元素》(*Fifth Element*)、《完美风暴》(*The Perfect Storm*)、*X2 XMen United*、《海底总动员》*Finding Nemo* 等等的电影中都应用了这个技术。在很大程度上,这些作品中所使用的算法都运用了快速傅立叶变换(Fast Fourier Transform, 简称FFT)对随机噪音做了小心的处理,此随机噪音作为一个随着时间跟频率相关的周相位移[Tessendorf02]。那些相同的算法不需做出太大的调整就可以应用到游戏代码中[Jensen01]、[Arete03],并且可以制作出美丽的海洋表面,在普通的硬件中一般能达到每秒30帧,可能会更多。

然而,FFT算法不能实现物体和水体表面的交互。要做到这个还是困难重重的,例如,让角色趟过一条溪流并产生一些搅动,而这些搅动直接取决于玩家所控制的动作;水上摩托在水面上行驶时并不会引起惊涛骇浪。使用基于FFT的模拟,浴缸中的水波不会来回地荡漾。总的来说,使用FFT的方法把任意的物体放入水中,像在真实世界中一样使之与水面产生互动,同时又不造成帧的损失,这是不可能的。从实用的目的来说,波浪表面的模拟还受到一些限制,如高度数据可以在一帧内或帧之间进行修改等。

这篇文章提供了一种称作iWave的新方法,它用来计算水面波浪的传播,并且可以突破上述限制。溪流、水上摩托和浴缸这三种场景都可以使用iWave处理得十分出色。任何形状的物体都能放在水面并能产生波浪。实际上,涌向物体的波浪碰到物体后会反弹回去。整个iWave的算法最终归结到二维卷积和掩码这两种适合使用硬件加速的操作。甚至,即使没有硬件的支持,纯软件的实现也能在1GHz处理器上以30FPS的速度模拟 128×128 的水面高度的网格。较大一些的网格理所当然会使帧速率减慢,而较小一些的网格则可以使之加速;速度和网格点的数量是直接成比例的。由于这种方法避免了FFT,因此它的互动性很强,应用范围很广。

3.6.1 线性的波浪

在进入主题之前，我们先快速回顾一下水体表面波浪运动的方程式。[Kinsman84] 是一个非常棒的有关流体动力学详细资料的资源。适用于此处的公式称作线性伯努利方程 (Linearized Bernoulli's Equation)。我们在这里所使用的方程采用了一个非常奇怪的操作符，等一下我们会解释它。在 [Tessendorf02] 一文中给出了方程如下：

$$\frac{\partial^2 h(x, y, t)}{\partial t^2} + \alpha \frac{\partial h(x, y, t)}{\partial t} = -g \sqrt{-\nabla^2} h(x, y, t) \tag{3.6.1}$$

其中， $h(x, y, t)$ 是水面的高度，它的含义是，在时间 t 时，水平位置 (x, y) 处的水面高度。左边的第一项是波浪的垂直加速度。左边的第二项，带有常数 α 的，则是速度的阻尼项，它不总是水面波浪方程的一部分，但是有时对控制可能发生的数值不稳定性很有帮助。右边的项是从质量守恒和重力恢复力结合而来。这个操作符

$$\sqrt{-\nabla^2} \equiv \sqrt{-\frac{\partial^2}{\partial x^2} - \frac{\partial^2}{\partial y^2}}$$

是质量守恒操作符，我们认为它是表面的垂直导数。它的作用就是保持被移动水的总质量守恒。当一个地点水面高度上升的时候，在这个位置上的水的质量将会增加。要使质量守恒，水面必有某个地方高度下降，其减少的水量与前一个水面升高地点增加的水量是相同的。

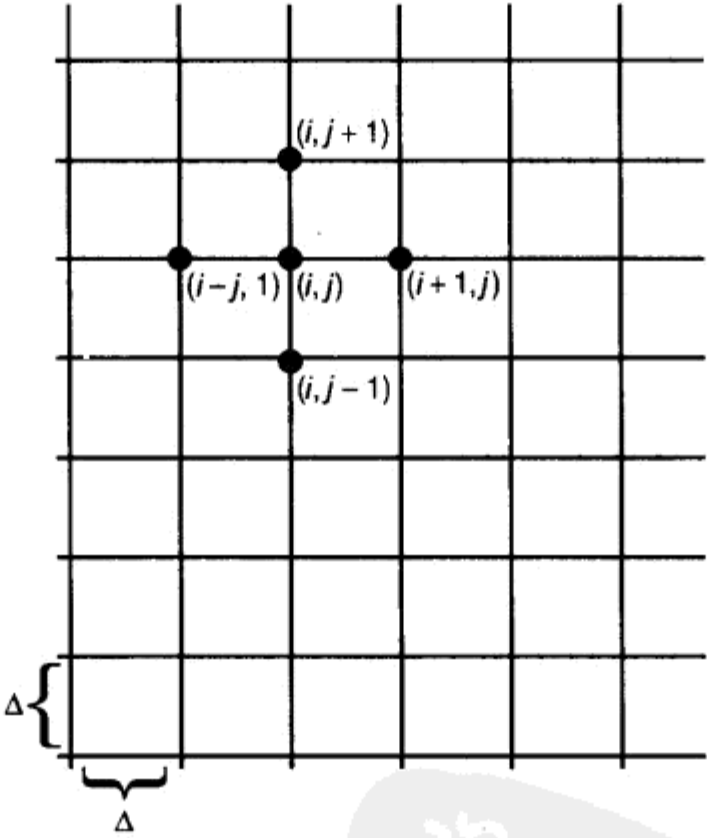


图 3.6.1 计算波浪高度的网格布局

下一节描述的是如何通过把方程表示成卷积来求方程 3.6.1 右边的值。在本文的剩余部分中，高度都是在一个规则的网格上计算的，如图 3.6.1 所示。水平位置 (x, y) 变成了网格位置 (i, j) ，在该位置 $x_i = i \Delta$ ， $y_j = j \Delta$ ，在网格的两个方向上的间隙都为 Δ 。下标 $i = 1, \dots, N$ ， $j = 1, \dots, M$ 。

3.6.2 垂直导数操作符

和所有作用于一个功能的线性操作符 (Operator) 一样, 使用垂直导数的时候, 可以把它当作它所应用的功能卷积。在这一节中, 我们就要建立这个在正常的网格上应用高度数据的卷积。同时, 我们也要得出卷积的最佳大小、计算拍打的权重。

在一个卷积中, 垂直导数在有高度网格上的操作是

$$\sqrt{-\nabla^2} h(i, j) = \sum_{k=-P}^P \sum_{l=-P}^P G(k, l) h(i+k, j+l) \quad (3.6.2)$$

卷积核是大小为 $(2P+1) \times (2P+1)$ 的正方形, 它可以预先被计算出来, 在模拟开始之前存储在查询表中。对于核心大小 P 的选择同时影响了速度以及模拟的视觉质量。如果要想清楚地得到像水一样的动态, 那么最小值是 $P=6$ 。

图 3.6.2 表现了以 k 为参数的核心要素函数 $G(k, 0)$ 。两条垂直的虚线在 $k=6$ 和 $k=-6$ 两点上。从图中可以看出, 当 k 的值大一些的时候, 函数值几乎为零, 虚线外 k 值对卷积的影响不大。如果 k 值再小一些的时候, 如 $k=5$ 和 $k=-5$ 时我们就停止卷积, 测算卷积的速度会加快, 但是我们会错过当 $k=6$, $k=-6$ 时出现的一些小变化。经验显示, 如果使这个值达到 6, 可以得到非常好看的波浪; 但是如果有计算时间方面的压力时, 卷积可以就此打住, 只是在视觉上不那么真实。在 $|k|<6$ 的时候就终止卷积是以牺牲大量波动为代价的。这也分析了为什么 $P=6$ 被认为是合理的波浪模拟最佳折衷选择。

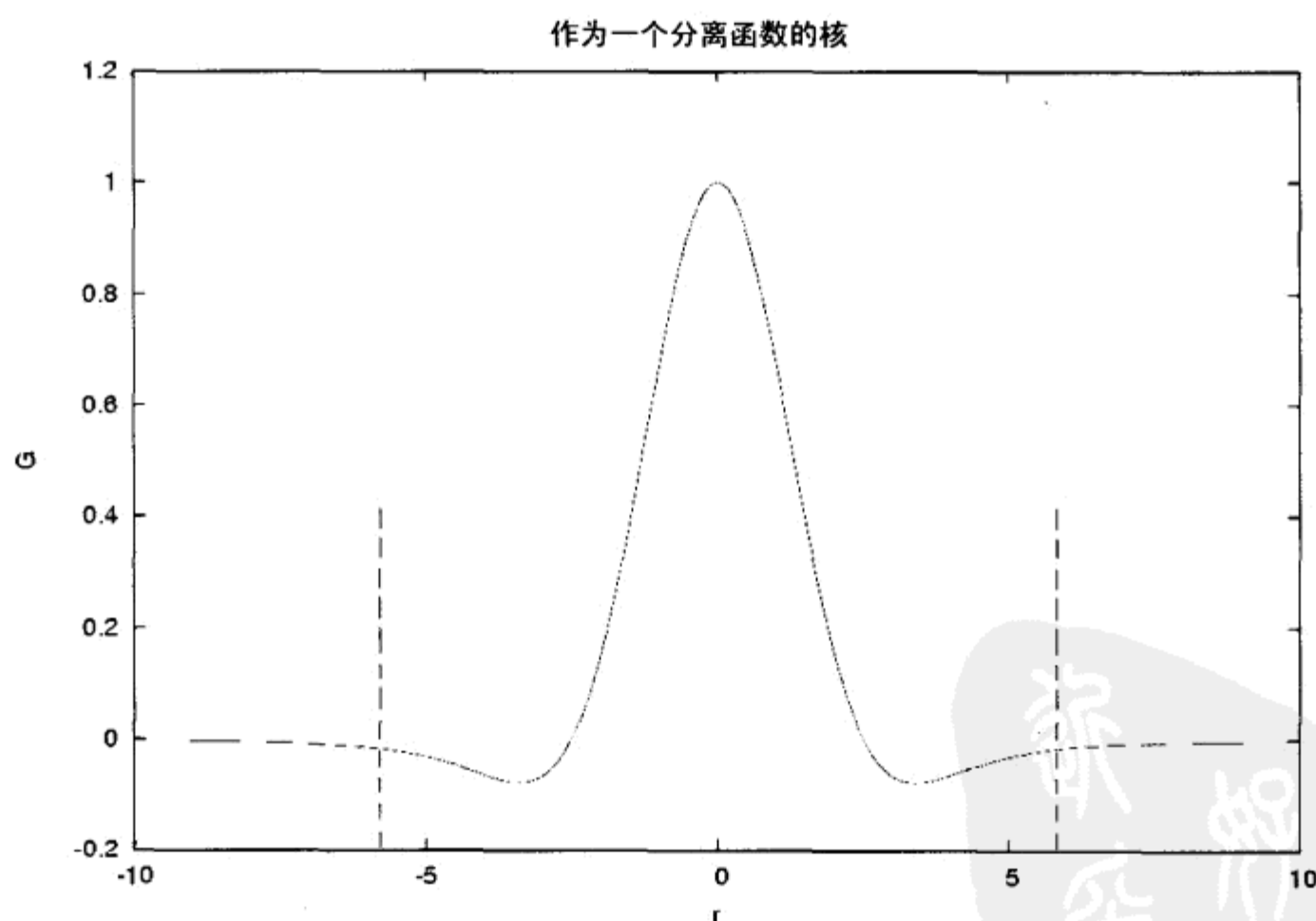


图 3.6.2 横截面中的垂直导数核心。在两个虚线中间的部分是 $|k|<6$

计算核心函数值并把它们存储在一个查询表中是一个相对简单直观的过程。第一步是计算一个单个数字，它会调整中心值为一的核心值。这个数字是

$$G_0 = \sum_n q_n^2 \exp(-\sigma q_n^2)$$

这个求和计算中， $q_n = n \Delta q$ 中选择 $\Delta q = 0.001$ 有利于准确性，而 $n = 1, \dots, 10\,000$ 。因数 σ 使求和结果接近一个合理的数字， $\sigma = 1$ 是个不错的选择。有了这个数字，当 $r = \sqrt{k^2 + l^2}$ 时，核心函数值是

$$G(k,l) = \sum_n q_n^2 \exp(-\sigma q_n^2) J_0(q_n r) / G_0$$

计算所需要素的时间相对较小，全部成本只是一次初始化；一旦这些要素计算出来了，在模拟中就固定了。

剩下的需要计算卷积核心要素的一项是一个用于 Bessel 函数 $J_0(x)$ 的公式。它已经包括在 C 标准数学函数库中，就是 `j0()`。如果你没法取得这个函数，[Abramowitz72] 中提供了一个很方便的近似办法。虽然那里面的方程需要合适的参数形式，但在单精度需要的范围内还是准确的，也能很好地达到模拟目的。

当在每一时长做卷积操作的时候，是有机会优化其速度的，不管是在某个硬件配置中还是在软件中进行。由于卷积函数核心中的两个对称性，那么软件的优化紧跟其后：函数核心是旋转对称的， $G(k,l) = G(l,k)$ ，并且函数核心也是相对两条轴 $G(k,l) = G(-k,-l) = G(k,-l) = G(-k,l)$ 的反射对称的。如果不使用任何对称性的优化，而直接计算方程 3.6.2 的卷积操作需要 $(2P + 1)^2$ 次的乘法及加法运算。应用这些对称性的特点进行优化后，卷积操作可以重写为（在构建时运用 $G(0,0) = 1$ 的条件）

$$h(i,j) + \sum_{k=0}^P \sum_{l=k+1}^P G(k,l) \left(\begin{aligned} &h(i+k,j+l) + h(i-k,j-l) + h(i+k,j-l) \\ &+ h(i-k,j+l) \end{aligned} \right)$$

这里的格式虽然仍然有 $(2P + 1)^2$ 次的加法运算，但是只有 $(P + 1)P / 2$ 次的乘法运算。这种卷积操作可以被设计成一种适用于 SIMD 管线的计算格式，因此显卡及 DSP 都能有效地执行卷积操作。

3.6.3 波浪的传播

既然我们可以测算出有高度网格上的垂直导数，水面的传播就可以随着时间的推移计算出来了。最简单的就是在每个步长内运用一个显性方案。虽然复杂一些的隐性方法会更准确更稳定，但同时也比较慢。鉴于我们在这篇文章中要解决的是线性方程式，则显性的方法在有摩擦的情况下速度快而且稳定，而时长的大小可以根据显示的帧速率的需要随意设定。虽然从游戏角度出发，如果没有任何波动源存在时波浪能慢慢消散，但是如果需要的话，摩擦可以保持得很低。

要得到显性的解决方法，方程 3.6.1 中的时间导数必须写成有限微分形式。第二个导数项可以通过对称微分来建立，而用来减弱波浪的摩擦系数项可以当作一个前项微分。假设一个时长为 Δt ，重新整理这些结果项后，则下一个时长的有高度网格是

$$h(i, j, t + \Delta t) = h(i, j, t) \frac{2 - \alpha \Delta t}{1 + \alpha \Delta t} - h(i, j, t - \Delta t) \frac{1}{1 + \alpha \Delta t} - \frac{g \Delta t^2}{1 + \alpha \Delta t} \sum_{k=-P}^P \sum_{l=-P}^P G(k, l) h(i + k, j + l, t) \quad (3.6.3)$$

就数据的结构来说, 这个传播算法可以在三组拷贝的高度域网格中运行。要讨论这个, 这三组网格拷贝被存储在浮点数组 `height`、`vertical_derivative` 和 `previous_height` 中。在模拟过程中, 数组 `height` 总是存储着最新的有高度网格, `previous_height` 存储着前一时长的有高度网格, `vertical_derivative` 存储着前一时长的有高度网格的垂直导数。在模拟开始之前, 这些要素应该全部初始为零。以下是完整的传播过程的伪代码。

```
float height[N*M];
float vertical_derivative[N*M];
float previous_height[N*M];

// ... 初始化为零 ...

// ... 开始帧循环...

// --- 这是一段波浪传播代码 ---
// 使用核对高度进行卷积, 然后把结果放入垂直方向导数
Convolve( height, vertical_derivative );

float temp;
for(int k=0; k<N*M; k++)
{
    temp = height[k];
    height[k] = height[k]*(2.0-
alpha*dt)/(1.0+alpha*dt)
- previous_height[k]/(1.0+alpha*dt)
- vertical_derivative[k]
*g*dt*dt/(1.0+alpha*dt);
    previous_height[k] = temp;
}
// --- 波浪传播代码结束---

// ... 帧循环结束...
```

`vertical_derivative` 和 `previous_height` 的数量可以用来修饰波浪的视觉效果。例如, 如果 `vertical_derivative` 是一个大数值, 那么显示有很大的重力把波浪吸引回平均位置。在 `vertical_derivative` 数值大的点上, 对 `previous_height` 值和 `height` 值作比较, 可以大致判断出波浪是处于波峰还是波谷。如果是在波峰, 在这一区域可以使用显示泡沫的材质。这并非一种用于物理或海洋学的具体算法, 而只是关于如何找到波浪最高点的想法。研究它的意义在于, 两个额外的网格, `vertical_derivative` 和 `previous_height`, 除了只对传播步骤有用以外, 可能对模拟以及波浪区域高度的渲染也能发挥作用。

3.6.4 可以互动的障碍物及其发生源

截至此处，我们已经建立了一个模拟水体表面波浪传播的方法。这个方法包含一种相对较快的卷积操作，我们讨论过的所有东西都有可能完成，就像在前面介绍中谈到的那种效率较高的 FFT 方法那样。这种卷积操作的关键是它的简便，有了它，只需一些额外的二维处理就可以产生水中物体与水之间真实性非常强的互动，以及当水被抽走时表面的搅动。

我们通过二维处理就制造出互动，从某种角度来说是个奇迹。一般情况下，在流体动力学的模拟中，流体在边界或者靠近边界的速度是根据边界条件的类型重设的，并需要清楚地了解边界几何形状，如它的外向法线。现在，我们没有做任何分析就做出了模拟，而这些方法对速度至关重要。

1. 来源

制造流体的运动的方法之一就是找到发生源。发生源可以表示为一个与有高度网格同样大小同样维数的二维网格 $s(i, j)$ 。发生源的网格应该在没有运动的地方数值为零。在波浪被“戳”和/或被“拉”的地方，发生源的网格的数值可以为正或者是负。然后，就在公式 3.6.3 中计算传播步骤之前，有高度网格进行 $h(i, j) = h(i, j) + s(i, j)$ 更新。发生源由于是在每帧中对环境进行能量的输入，它在模拟过程中应该被改变，除非我们需要恒定的能量增加。一个波动源可以产生水面上的波纹。

2. 障碍物

在这一方案中对于障碍物的实施非常简单。对于一个额外的障碍物网格填入浮点值，主要使用两个极值。这个网格就像掩码一样，指出哪里存在着阻碍。在每个网格点，如果不存在障碍物，那么那一点障碍物网格的值为 1.0。如果网格点被障碍物所占，那么障碍物 grid 的值为 0.0。障碍物周围边界上的网格点上，障碍物网格的值介于 0.0 和 1.0 之间。而中间区域就是为了防止障碍物边缘出现锯齿。

基于这个障碍物掩码，计算障碍物的影响就是简单地把有高度网格和障碍物掩码相乘，这样波浪的高度在障碍物存在的时候被迫为零，并在障碍物外面的区域保持不变。令人惊讶的是，这就是妥善解决水体表面物体需要做的全部！这样简单的一个步骤就能造出向障碍物传播并正确反弹的波浪。它还可以造出通过障碍物狭长缺口的波浪。并且允许障碍物为任何形状，使用者想怎样就怎样。一个有发生源和障碍物的应用的伪代码是：

```
float source[N*M], obstruction[N*M];  
// ... 设置发生源和障碍物  
  
for(int k = 0; k < N*M; k++)  
{  
    height[k] += source[k];  
    height[k] *= obstruction[k];  
}
```



```
// ... 现在开始波浪传播计算
```

3. 尾迹

由移动物体产生的尾迹是通过互动的 iWave 方法自然制造的。在这个特殊的例子中，障碍物的形状也是发生源的形状。只要在障碍物边缘的周围有一个防止锯齿区域，设置 `source[k] = 1.0 - obstruction[k]` 就可以了。在这种选择下，在网格中移动一个障碍物能制造出障碍物过后的尾迹，包括 V 型的 Kelvi 尾迹。它还能制造出一种船尾迹波纹以及沿着障碍物边沿流动的波浪。障碍物的形状和运动都对形状、时间，以及尾迹波其他一些扩展部分的细节有很大影响。

3.6.5 环境波浪

iWave 方法对于生成持久的大规模波浪现象，如开放的海洋波浪，并不十分有效。如果在一个应用中希望的是这种不由 iWave 方法生成的“环境波浪”，那么就有一个附加的程序，以防止环境波浪的简单模拟。

环境波浪包含已经由其他一些步骤生成的有高度网格。例如，FFT 方法可以用来生成海洋波浪，并把其存储在有高度网格中。由于我们只是想计算环境波浪与障碍物的互动，环境波浪对模拟的影响应该保持在障碍物区域以内。应该在完成的波浪传播之前，在计算障碍物和发生源一起作用之后来应用这个方法修改高度网格。这个方法的伪代码如下。

```
float ambient[N*M];

// ... 设置当前环境网格

// ... 就在发生源和障碍物代码后，进行：
for (int k = 0; k < N*M; k++)
{
    height[k] -= ambient[k]*(1.0-obstruction[k]);
}

// ... 现在开始波浪传播计算
```

有了这个方法，不管何种特性的环境波浪都可以和任何形状的物体发生互动。

3.6.6 网格的边界

到此为止，我们一直都回避着如何处理网格的边界的问题。问题就出在，卷积函数核需要数据从网格点上获得一些数据，一个 4 个方向上的距离 P ，即卷积中心网格点到临近网格点的距离。所以，当中心网格点少于从 P 点到网格的边界，丢失的数据必须要根据某些标准生成。有两种边界条件应用起来很容易，周期边界和反射边界。

1. 周期边界

在这种情况下，波浪遇到边界看上去要向边界的另外一侧继续传播。在边界附近发生

卷积的情况下, 方程 3.6.3 ($i+k$ 和 $j+l$) 中的网格坐标有可能是在 $[0, N-1]$ 和 $[0, M-1]$ 的范围之外的。使用模数 $(i+k) \% N$ 就能保证处于 $[-N+1, N-1]$ 范围以内。为了保证结果总为正, 我们可使用一个双模数: $((i+k) \% N + N) \% N$ 。对 $j+l$ 坐标进行同样处理保证了强制运用周期边界条件。

2. 反射边界

反射边界可以使波浪转头, 并从波浪碰撞边界传回到网格, 很像遇到水中障碍物反射回去的波浪一样。如果坐标 $i+k$ 大于 $N-1$, 那么它会变为 $2N-i-k$ 。如果坐标小于零, 它会变号; 也就是说, 它会变为 $-i-k$, 而这个是正值。对 $j+l$ 坐标需要运用一个同样的程序。

要有效地运用这两种处理边界, 最快的方法就是把网格分为以下 9 个区域:

- (1) 坐标的范围是 $i \in [P, N-1-P]$ 到 $j \in [P, M-1-P]$ 的网格的内部部分;
- (2) 右手边的是 $i \in [N-P, N-1]$ 和 $j \in [P, M-1-P]$;
- (3) 左手边的是 $i \in [0, P-1]$ 和 $j \in [P, M-1-P]$;
- (4) 顶边的是 $i \in [P, N-1-P]$ 和 $j \in [0, P-1]$;
- (5) 底边的是 $i \in [P, N-1-P]$ 和 $j \in [M-P, M-1]$ 。
- (6) 剩下的 4 个角。

在每个区域内, 在没有条件或者其他一些模数操作下, 特殊的边界处理能有效地进行编码。

3.6.7 表面张力

在这之前, 我们讨论的模拟都是受重力影响的波浪传播。受重力作用的波浪影响着水面的流动, 其范围大概是一英尺或多一些。在小一些的范围, 传播的特性发生了改变, 包括表面张力。表面张力能使波浪在更小的空间范围内传播得更快, 这会使水面比不存在表面张力时显得硬实些。就我们的目的来说, 表面张力的特性就是长度比例系数 L_T , 它决定表面张力波浪的最大尺寸。我们的程序要求的惟一变动就是卷积函数核计算不同。卷积函数核的计算变为

$$G(k, l) = \sum_n q_n^2 \sqrt{1 + q_n^2 L_T^2} \exp(-\sigma q_n^2) J_0(q_n r) / G_0$$

除了这个变动以外, 整个 iWave 处理方法是相同的。

3.6.8 结论

用于水面传播的 iWave 方法是一种很灵活的生成可互动扰乱水面波动的方法。由于它是建立在二维卷积和一些简单的二维图像处理基础上的, 即使在只有软件加速计算应用中也能达到很高的帧速率。卷积的硬件加速计算可以使 iWave 适用于多种游戏平台。游戏中水面与物体间互动的增加可以开创游戏开发者从前不可能到达的游戏境地。

3.6.9 参考文献

[Abramowitz72] Abramowitz, Milton, and Irene A. Stegun, *Handbook of Mathematical*

Functions, Dover, 1972. Sections 9.4.1 and 9.4.3.

[Arete03] Arete Entertainment, available online at www.areteis.com.

[Jensen01] Jensen, Lasse, online tutorial, available online at www.gamasutra.com/gdce/jensen/jensen_01.htm, 2001.

[Kinsman84] Kinsman, Blair, *Wind Waves*, Dover, 1984.

[Tessendorf02] Tessendorf, Jerry, "Simulating Ocean Water," *Simulating Nature*, SIGGRAPH Course Notes, available online at <http://users.adelphia.net/~tessendorf/>, 2002.



3.7 用多层物理模拟快速变形

作者: Thomas Di Giacomo 和 Nadia Magnenat-Thalmann,
MIRALab, C.U.I., University of Geneva (日内瓦大学·瑞士)
E-mail: thomas@miralab.unige.ch, thalmann@miralab.unige.ch
译者: 李鸣渤
审校: 刘永静

游戏的一个主要目的就是为游戏者提供高水平的交互。变形, 例如令人信服的布料动画、带皮肤的面部动画、柔软的身体等等, 都能够给游戏者带来更多的体验, 增强游戏的吸引力。然而, 变形计算起来成本较高, 这主要是因为其中需要众多的物理因素, 还有就是对碰撞的处理。虽然最近的一些著作提出了各种可以加快变形计算的方法, 但是游戏对可变形物体的使用依然甚少。把变形与其他模块及资源结合起来, 包含在一个完整的游戏引擎中仍然是不可取的, 而且基于物理的变形实现仍然是相当复杂的。

这篇文章介绍了一个简单而有效的创建变形的办法。这种方法不仅计算成本低, 而且制作出来的变形效果好、容易控制, 可以为您的游戏环境增添许多生气。它的主要概念就是把在柔体模拟中广泛使用的质量块弹簧系统简化为一个运行速度更快的两层 (two-layer) 问题。这里所述的动画系统可以很好地运用于不同粗细的柔软线形可变形的物体 (体积可变的电线、管道、蛇等等), 它还可以通过在预先建模步骤中调整两个质量块弹簧层的拓扑结构, 使其可以很容易在其他物体上的运用, 如皮肤的变形 (和 [Jianhua94] 很相似)、衣服袖子。

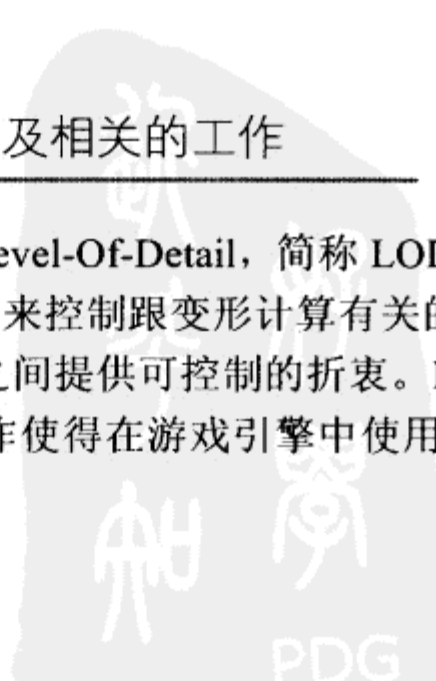


把这样的可变形物体结合到您的游戏环境中是十分直观的。
ON THE CD 随书附送的光碟中就包含了可以在普通电脑中操作的例程代码。

此外, 在文章中还介绍了把它方便地移植到手持 PDA 及可编程 GPU 的详细方法。

3.7.1 基于物理的动画 LOD 及相关的工作

与渲染几何体的精细度 (Level-Of-Detail, 简称 LOD) 相似, 我们可以为基于物理的动画使用 LOD 来控制跟变形计算有关的昂贵的 CPU 和存储花费, 并在速度和真实性之间提供可控制的折衷。LOD 已经完成的以及正在进行的物理方面的工作使得在游戏引擎中使用实时物理变得更加可行。



1. 物理 LOD

要提高基于物理的动画引擎速度，主要有两种方法。第一种是优化计算成本昂贵的步骤，即碰撞检测和对时间的积分。第二种是在动画中使用 LOD。就渲染过的几何体而言，动画是可以被提升并且可以在一定程度上被简化的。例如，像在 [Berka97] 中所描述的那样当动作对于人的视觉而言太快、太远、或太多的时候，或者当动作不值得计算或不需要复杂计算的时候。为了简化动作或优化动作，我们需要建立一个可修改的动画引擎，以确保在不同 LOD 级别间平稳过渡（以避免 LOD 转换时出现间歇性动画）。例如，您可以通过降低动作的取样频率以及人体关节的自由度，将此方法应用在有关节的人体上 [Granieri95]。通过在一些 LOD 级别中使用基于物理的动画，在另一些级别中使用程序化的动画，你就可以建立混合模型 (Hybrid model)。或者，你可以建立一个单一多分辨率动画技术，它在每个 LOD 级别中都使用相同的技术。使用这样一个动画引擎，在游戏过程中你可以随时根据所需的视觉要求、有效的系统资源、所需表现等来选择合适的 LOD。显而易见，在过渡中，多分辨率模型和混合模型相比，能提供更简单的动作连贯性。另一方面，对于动画，混合模型可以提供的复杂程度及行为的范围更广。

2. 混合模型

在动画引擎中，混合模型将程序化动画和基于物理的动画组合成不同的级别，用来支持 LOD。我们参考 [Carlson97] 中的一个例子，它模拟了一个具有三种不同的分辨率的单腿机器人，按照从最复杂最真实到最快速的可分为：动力学的、运动学的，及单质点运动。在这些级别上的过渡只可能在特定的时间实现，即当物体达到特定状态的时候。如果你想在游戏中出现自然环境的动画的话，请参阅 [Perbet01] “实时运动的草原 (*Animating Prairies in Real-Time*)”，或者 [Di Giacomo03] “树的实时动画 (*Real-Time Animation of Trees*)”，它使用了一个类似的 LOD 理念。这两位作者都通过动作（根据所需的细腻程度）的立体或线性混合，确保不同层次间的平稳过渡。有些混合系统在快速变形中使用多层质量块弹簧系统。例如，为了模拟衣物上的褶皱，[Kang02] 将两个质量块弹簧系统“循环连接” (loop-linked) 组合起来：一个粗略的模型用于整体的基于物理的变形，一个密集模型用于小的几何体变形，这样就产生了褶皱。[D'Aulignac99] 提出了另一个多层质量块弹簧系统的例子，它常常被用来模拟人类模型的变形。

为了使混合模型发挥作用，不同的 LOD 方法必须有能够随着 LOD 进行适当的提高或降低的资源成本。此外，不同的方法应该受到限制，以便在相邻的 LOD 中，结果能够相互匹配，确保平稳过渡。

3. 多分辨率 LOD 模型

现有的应用在物理 LOD 上的多分辨率模型主要使用质量块弹簧及有限元网络 (finite element network)。大致上，这些模型都是在局部即时动态地优化或简化物理节点的网格模型。例如，[Hutchinson96] 优化了当某种约束被强制应用时的质量块弹簧系统（如，当弹簧间的角度超过了一个临界值时）。[Brown01] 使用一个简化过的质量块弹簧系统以减少计算。通过对特定应用中（如外科手术中的切开）变形的位置加以利用，物理上的计算就被限制在当

前位置的几个控制点上, 所得到的变形也延伸到其他节点。在实时的应用中, 有限元的方法可以通过改进后的技术进行加速, 比如 [Debunne01] 提出的运用过程网状模型和适合的时长, 以及 [Capell02] 提出的运用多分辨率分级体积细分过程 (multiresolution hierarchical volumetric subdivision) 来模拟动态变形。只要有创造力, 在加上一点时间, 你就很有可能把所有引用过的资料融入你的游戏引擎中, 使你能够创造出更加生动更有互动性的游戏世界。

3.7.2 使用分层的质量块弹簧物理的快速变形

运行速度是个主要的问题; 你必须减少用于计算动力学的点的数量。同样地, 这个基于物理的动画引擎也有三个主要特征。首先, 用于物理的网状模型包含的顶点和面远比相应的被渲染网状模型要少。其次, 变形的模拟是通过一个双层的质量块弹簧网络完成的, 其中第一层被看作是一个中枢结构, 它控制着全局的运动; 而第二层只在单维中运动 (以限制计算量)。后一层控制着包迹及物体大致的体积变形, 例如, 由场景中的物体间碰撞产生的压缩。有一个简单的方法可以理解两个层次组合的作用, 就是把整个变形体看作一个普通的圆柱体。中枢层计算延伸、扭转, 以及圆柱体轴的转动。而第二层则计算局部的受限制的放射性变形。最后, 使用 Verlet 积分器为系统提供速度及稳定性。分层的质量块弹簧系统能为游戏提供完美的变形, 同时在计算成本上则十分低廉。

1. 中枢层的表示及模拟

在这个方法中, 线性的质量块弹簧网络就是的中枢结构, 它使物体的“骨架”做出延伸、扭转及转动的动作 (见图 3.7.1)。轴状的中枢结构还能为体积变形层提供几何输入和限制。

中枢是由用线性弹簧连接的质量块构成的, 每个质量块只有一个质量父体 (parent) 和一个质量子体 (child), 末端除外。至少有一个质量块是没有父体的, 它存在于网络的一端, 我们称之为根质量块 (root mass)。

中枢运动的模拟是很直观的。我们必须通过在一维网络中应用的 Hooke 定律为每个质量块计算力 f_i :

$$\vec{f}^i = -k_s^i \frac{\vec{x}^i - \vec{x}^{i-1} - l_0^i}{\|\vec{x}^i - \vec{x}^{i-1}\|} + k_s^{i+1} \frac{\vec{x}^{i+1} - \vec{x}^i - l_0^{i+1}}{\|\vec{x}^{i+1} - \vec{x}^i\|} \quad (3.7.1)$$

其中, k_s 是第 i 个弹簧的弹性系数, l_0 是第 i 个弹簧未拉伸时的长度 (均为常量); x_i 是第 i 个质量块的位置。通过中止根质量块上的外力及内力, 使用者可以调整它的位置, 如与鼠标的互动。同样, 根质量块可以受同一场景内另一个物体的在几何上限制; 例如, 受到一个虚拟人物手的限制。你可以根据游戏所需的行为增加与根节点相似的额外固定节点。一旦这些力计算出来了, 就可以运用一个 Verlet 积分器随着时间前移来更新中枢的形状了。

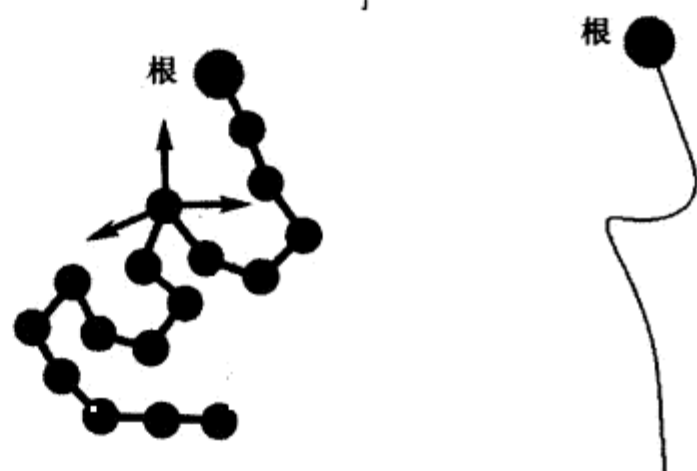


图 3.7.1 中枢弹簧系统 (左图), 带有足够的质量块和合适的弹簧长度的结果线 (右图), 顶部灰色的质量块就是根质量块, 它的运动由用户控制

2. 体积变形层

第二个质量块弹簧层用来模拟体积的变形。这一层是由一系列的交叉部分组成的，每一个交叉部分有4个放射型质量块。这些交叉部分代表了体积变形的包迹。要做出最佳的变形，虽然使用较少的交叉部分是允许的，但是我们还是应该为中枢层的每个质量块创建一个单独的交叉部分。体积包迹质量块是通过线性受限制的弹性弹簧与中枢质量块相连的。图 3.7.2(左图)表示了其中一个体积包迹交叉部分。

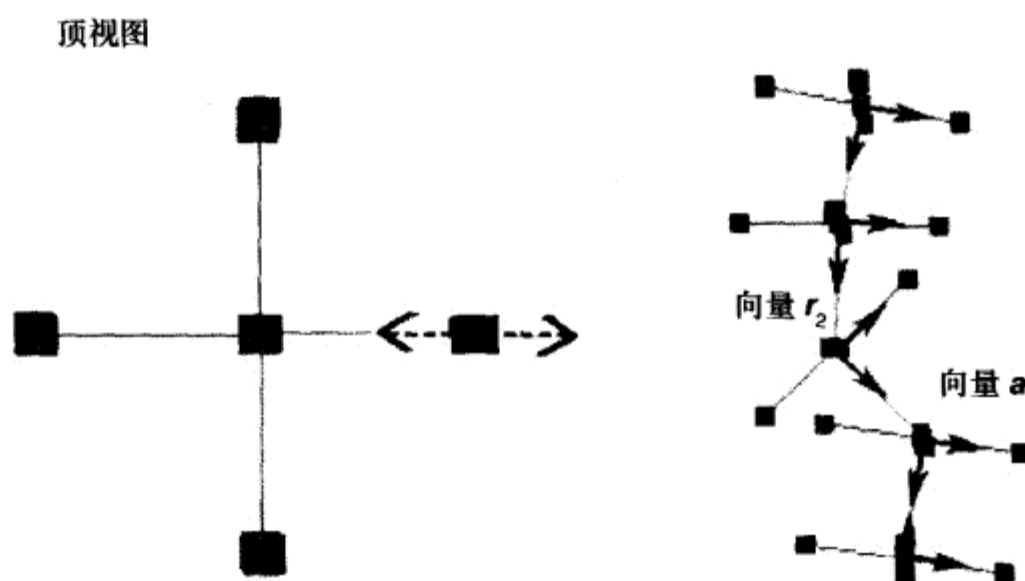


图 3.7.2 一维放射性的变形(左图), 在每一帧中, 取决于中枢结构的包迹半径的重新架构(右图)

如图 3.7.2 所示, 每个放射性质量块都受到了限制, 它们必须沿着自己的放射方向做单维移动。模拟体积变形所需的工作只比中枢模拟多一点点。由于是在一维(1D)的情况下, 因此动力学的计算大大减少; 然而, 所得出的运动也只有一维。为了确保变形是三维(3D)的, 你需要对放射做出以下更新。首先, 在每帧中, 每个交叉部分的放射向量 r_i 都要被重新计算, 使之与其相应的中枢弹簧方向 a 垂直, 如图 3.7.2(右)所示。这部分的计算是这样的:

```
void UpdateRadii(vec a, vec* r1, vec* r2, vec* r3, vec* r4)
{
    //发现一个坐标轴不共线的向量
    if (a->x!=0)
        r1 = {-a->y, a->x, a->z};
    else if (a->y!=0)
        r1 = {a->y, -a->x, a->z};
    else
        r1 = {a->z, a->x, -a->y};

    // 计算剩余 3 个向量
    CrossVector(a, r1, r2);
    // CopyVector(destination, source)
    CopyVector(r3, -r1); CopyVector(r4, -r2);
    Normalize(r1); Normalize(r2);
    Normalize(r3); Normalize(r4);
}
```

更新后的半径定义了交叉部分质量块所能移动的方向。要注意的是 4 个向量 r_i , 它们都

进行了规一化 (normalize)，以简化交叉部分质量块的物理更新。

一旦你计算出新的交叉部分半径，那么接下来的步骤就是用改写过的 Hooke's 定律计算每个交叉部分质量块的一维力。

$$f^i = -k_s^i(x^i - l_0^i) \frac{x^i \vec{r}^i}{\|x^i - l_0^i\|} \vec{r}^i$$

(3.7.2)

其中， i 的取值为 $[0, 3]$ ，它是当前交叉部分包迹质量块的索引； x^i 是与在径向矢量 r^i 上质量块 i 的位置相对应的实数； l_0 是 r 上未伸展的位置。力 f^i 与相邻的包迹质量块没有关系，因为包迹节点不与弹簧相连。还要注意，这个力只施加在交叉部分质量块上，而非施加在相应的中枢结构的父轴节点上。因此，虽然可以通过在 Verlet 积分中重新配置以更改轴节点的位置来减小体积的区别，但是仍然不能保持体积不变。

3. 使用 Verlet 积分更新变形

一旦定义了力，我们就可以通过中枢及包迹质量块在时间上的积分（牛顿第二运动定律）来更新它们的位置，从而更新变形。在基于物理特性的引擎中，积分是一个十分重要的步骤，因为它决定了稳定性和模拟的速度。因此，我们使用简单直接的欧拉积分或 Verlet 积分来计算新的位置。有关后一方法的详细介绍，请参阅本书 Nick Porcino 的相关文章。基本的 Verlet 积分方法是一个较少使用速度的方案，它是基于当前位置和前一位置，而非质点的速度：

$$\begin{cases} \vec{x}_{t+\Delta t} = \alpha \vec{x}_t - \beta \vec{x}_{t-\Delta t} + \vec{a} \Delta t^2 \\ \vec{x}_{t-\Delta t} = \vec{x}_t \end{cases}$$

(3.7.3)

其中 $x_{t+\Delta t}$ 为下一位置， x_t 为当前位置， $x_{t-\Delta t}$ 为前一位置， Δt 是一恒定时长， a 是加速度，而 α 和 β 是拖拽系数。例如，我们可以在一个没有拖拽的系统中设 $\alpha=2$ 及 $\beta=1$ ，或者在一个存在少量拖拽的系统中设 $\alpha=1.99$ 及 $\beta=0.99$ 。由于这种积分方法不涉及速度，因此不会出现位置与速度的异步，这样就保证了很高的系统稳定性。（见表 3.7.1，稳定值与造成系统崩溃的力的强度相关）。

表 3.7.1 两种简单直观的积分方法的比较，使用相同的时长，相同的质量块及弹簧数量，同样进行了有碰撞及无碰撞测试

积分方案	速度	稳定性
Verlet	36~40 fps	$> 10^6$
显性欧拉	37~40fps	1

4. 由物理模型生成的渲染网状模型

在游戏中，如果用于变形的物理模型不与渲染后的网状模型相连，那么它是不起作用的。在理想状态下，为了回避昂贵的表皮计算，渲染几何体与物理模型会直接联系起来。第二个质量块弹簧层的包迹质量块会被组织在一起，共同形成一个渲染的网络模型雏形，但是这个包迹太粗糙了，不能提供高质量的视觉效果。使用粗略的包迹，把其作为 NURBS 或细分表

面的控制网状模型，我们能够获得质量更高的渲染网状模型，而不需要单独创建一个美工生成的表皮，也不需要进行昂贵的逐帧表皮计算。例如，如果你把包迹质量块看作是基于 NURBS 的插值控制点，那么你就可以以低廉的成本，离线计算 NURBS 网格、在不工作的时候更新所生成的顶点。Hermite 插值是最理想的：它输入两个位置（如，包迹质量块的位置），和这些位置上的两个向量（如，相对应的半径）。

```
void CEnvNod::GeomNodesConstruction()
{
    // 在半径 ID 位置上的投影
    vec pos = 3DGlobalPosition();
    vec broPos = m_Bros->3DGlobalPosition();
    // 异数插值
    vec der = 1.5*broPos;
    vec broDer = -1.5*pos;

    CInterp in = CInterp(pos, der, broPos, broDer);
    CAlpha alp = CAlpha(0.f, 1.f);
    m_NurbsPts[0] = in(alp(0.f));
    // 插值，存储在节点的 m_NurbsPts
    for(int s = 1; s <= Accuracy; s++)
        m_NurbsPts[s] = in(alp((float)s/Accuracy));
}
```

注意，为了简单起见，质量块位置要投影在半径上。通过指定插值步骤，点的数量、多边形的数量都是可以调整的，从而可以对在渲染步骤中权衡速度和真实性进行控制（见图 3.7.3）。

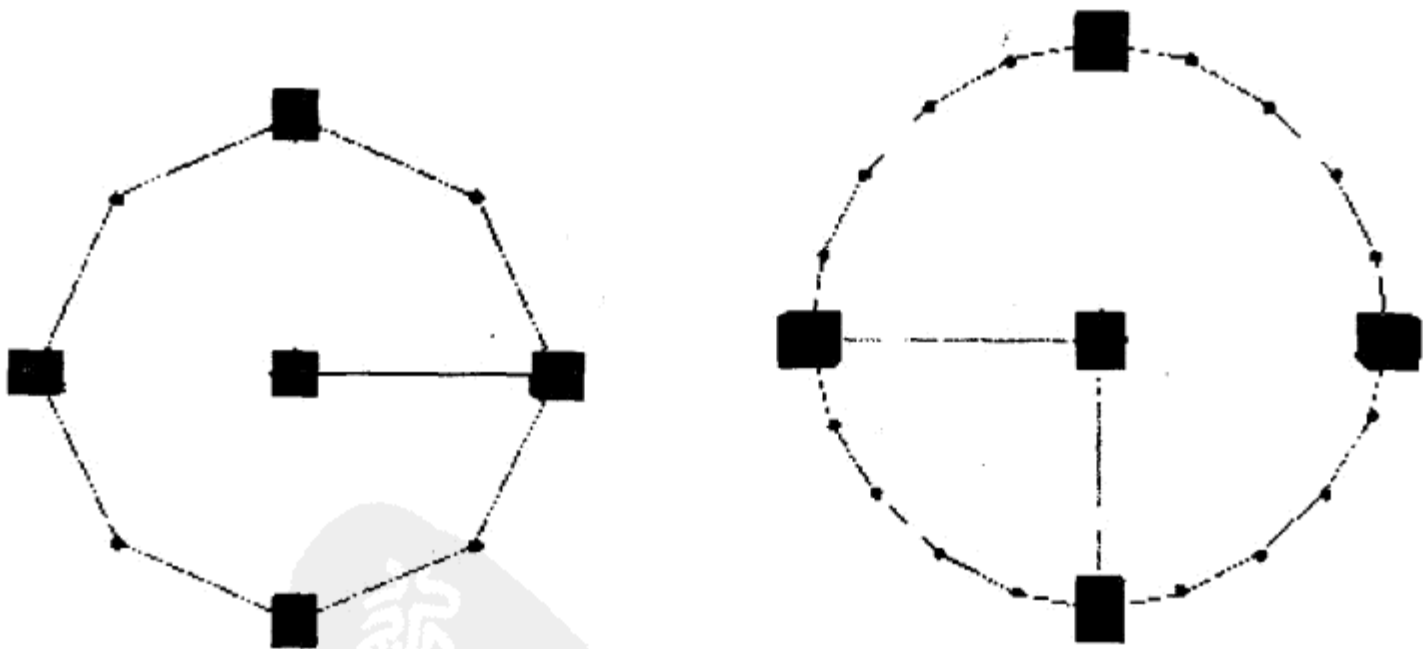


图 3.7.3 在相同的动画结构下，网格模型中的多边形数量可以根据 NURBS 细分的精度进行调整
(左图中是 2 个多边形，右图中是 5 个多边形)

使用这种方法计算几何点使我们能够根据包迹质量块的变形平滑地修改几何体，请参见图 3.7.4（左图）。然后，插值的几何节点沿着中枢层的方向连接到它们的垂直相邻部分以建立一组四方形，连同适当的法线用于渲染，参见图 3.7.4（右图）。

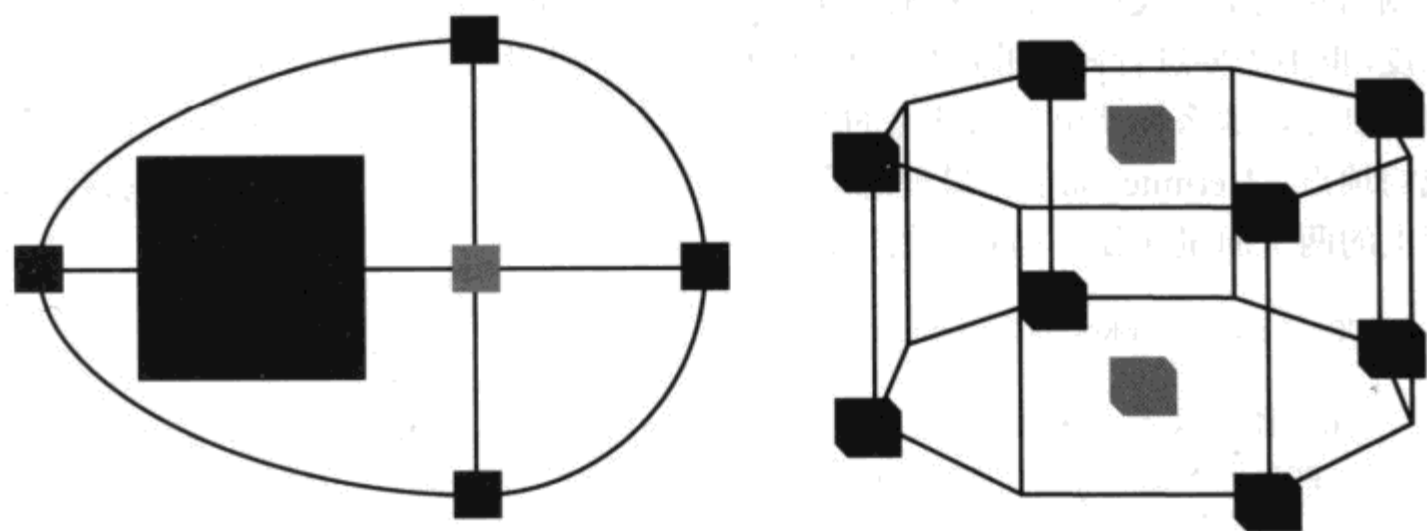


图 3.7.4 由物体和球形边界盒碰撞而产生的内部放射性变形的例子（左图），使用 NURBS 点的网格模型的结构（右图），灰色的是轴向质量块

5. 实例实现



随书附带的光碟中包含了这一技术在 PC 上应用的例程。为了使代码简单直接地不同平台上移植以及使用不同的渲染 API，我们把计算代码及渲染代码分开了。图 3.7.5 显示了使用 PC 代码的几种结果。

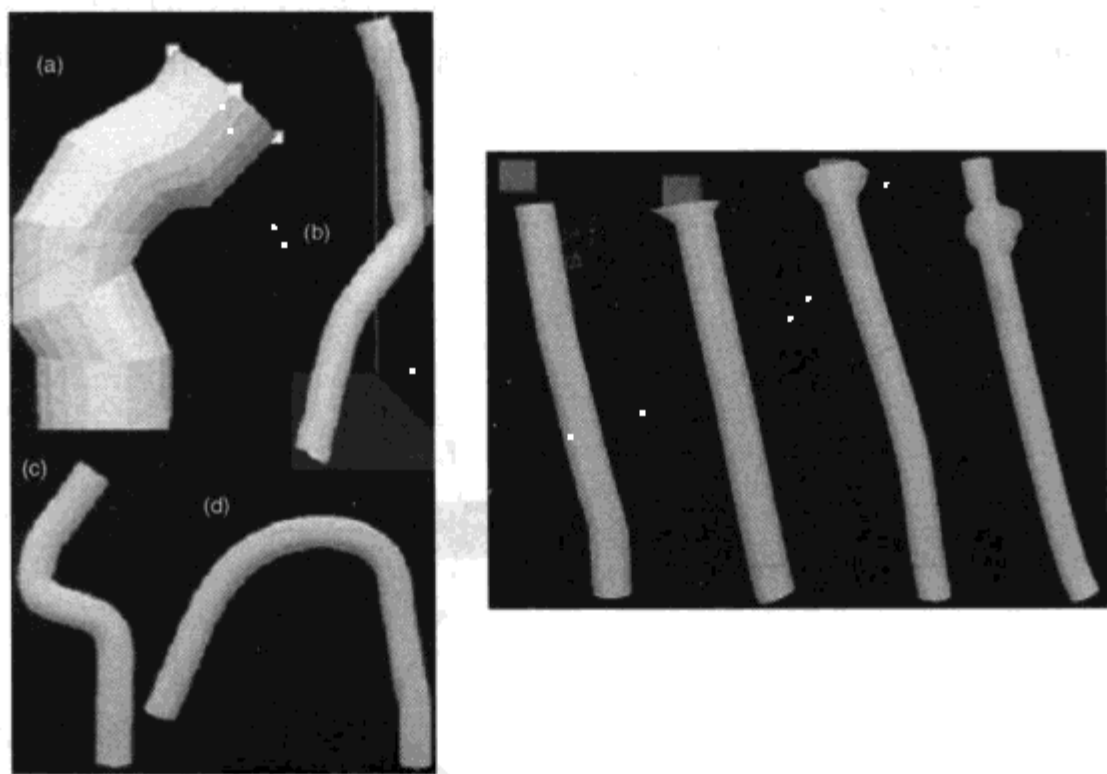


图 3.7.5 左边的图片显示了四边形的法线产生(a)，由于跟墙体发生碰撞而产生的包迹变形(b)，最终的结果(c)和(d)，右边的图片则显示了“内部”碰撞如何使模型变形的顺序。这些屏幕截图是取自在标准的 PC 上实现的即时动画

这一方法甚至还能移植到手持游戏机及 PDA 上。要开发如 PocketPC 的 PDA 上的图像显示部分，要注意几点情况。首先，这些装置目前还没有外部视频控制器（至少还没有实用性的）。其次，这些系统没有绘图管线，也不包含 VRAM，因此必须要使用主系统的 RAM 来存储软件的帧缓冲。以上两种所提到的代码中都使用系统内存，因而严重地限制了软件应用

的复杂性。最后，由于 PocketPC 缺少浮点处理器，而大多数图像数据都是浮点类型的，因此，在考虑如何在基于整数运算的硬件中运用基本数学时一定要谨慎。作者在 PocketPC 中对这一应用使用了一种简单明了的方法。代码对结果的计算类似于 PC 版本的方法，但是使用了浮点仿真运算。就是在渲染之前，把坐标转换成图形 API（这个例子中是 PocketGL 1.2）所能处理的范围的整数。

```
// 返回一个 PocketGL 整数型位置的函数
const vec* CIntPGLConversion::ConvertObjPosition()
{
    return ((int) iMaxPocketGL*m_vecPosition/fMaxGL);
}
// 每帧中调用的函数
void mainLoop()
{
    // 使用浮点数进行计算
    object->CheckWorldCollisions();
    object->ComputeForce();
    object->Integrate();
    // 转换成整数
    convertor->ConvertObjMesh(object->m_Mesh);
}
```

另一种可行的措施就是使用 GPU 硬件 shader 进一步加速动画。这一应用的几个子模块能在硬件中直接计算：半径的正交计算、Verlet 积分等等。

表 3.7.2 显示了一些在 PC 和 PocketPC 应用中的性能测试。准确性数值用于构筑几何体的细分的数目。

表 3.7.2 不同实现的性能

质量块的数目/NURBS 精度	PC 版本 (fps)	PocketPC 版本 (fps)
1 axial + 1 * 4 radial/1 accuracy	85	24
10 axial + 10 * 4 radial/2 accuracy	84	23
20 axial + 20 * 4 radial/4 accuracy	83	10
40 axial + 40 * 4 radial/6 accuracy	82	7
100 axial + 100 * 4 radial/6 accuracy	80	3
150 axial + 150 * 4 radial/6 accuracy	76	<1

3.7.3 结论

虽然从物理角度而言，应用这个方法的动画并不十分正确，但是运用它可以得到成本低廉、看似真实的变形，而且可以在各种游戏平台中很容易地实现。物理计算的减少主要是通过限制动态节点的数目，并把其中一些限制为一维来实现的。由于使用了双层化的方法，因此这个方法可以达到很好的效果。它使用粗略物理网格模型作为控制网格模型来生成渲染用的高分辨率网格模型，运用了弹簧之间的限制器，并使用 Verlet 积分方法来取得很好的稳定性。在不同平台的任何游戏引擎中集成这样的变形都十分简单直接。此外，CPU 和内存的使用都在可以接受的范围之内。

3.7.4 参考文献

[Berka97] Berka, Roman, "Reduction of Computations in Physic-Based Animation Using Level of Detail," *Proceedings of Spring Conference of Computer Graphics*, 1997.

[Brown01] Brown, Joel, et al., "Real-Time Simulation of Deformable Objects: Tools and Application," *Proceedings of Computer Animation*, 2001.

[Capell02] Capell, Steve, et al., "A Multiresolution Framework for Dynamic Deformations," *Proceedings of Symposium on Computer Animation*, 2002.

[Carlson97] Carlson, Deborah, et al., "Simulation Levels of Detail for Real-Time Animation," *Proceedings of Graphics Interface*, 1997.

[D'Aulignac99] D'Aulignac, Diego, et al., "Modeling the Dynamics of the Human Thigh for a Realistic Echographic Simulator with Force Feedback," *Proceedings of Conference on Medical Image Computing-Assisted Intervention*, 1999.

[Debunne01] Debunne, Gilles, et al., "Dynamic Real-Time Deformations Using Space & Time Adaptive Sampling," *Computer Graphics Proceedings (SIGGRAPH 2001)*.

[Di Giacomo03] Di Giacomo, Thomas, et al., "Real-Time Animation of Trees," *Graphics Programming Methods*, Charles River Media, 2003.

[Granieri95] Granieri, John, et al., "Production and Playback of Human Figure Motion for Visual Simulation," *Proceedings of Graphics Interface*, 1995.

[Hutchinson96] Hutchinson, David, et al., "Adaptive Refinement for Mass/Spring Simulations," *Proceedings of Eurographics Workshop on Computer Animation and Simulation*, 1996.

[Jianhua94] Jianhua, Shen, et al., "Human Skin Deformation from Cross Sections," *Proceedings of Computer Graphics International*, 1994.

[Kang02] Kang, Young-Min, et al., "Bilayered Approximate Integration for Rapid and Plausible Animation of Virtual Cloth with Realistic Wrinkles," *Proceedings of Computer Animation*, 2002.

[Perbet01] Perbet, Frank, et al., "Animating Prairies in Real-Time," *Proceedings of Symposium on Interactive 3D Graphics*, 2001.



3.8 快速且稳定的形变之模态分析

作者: James F. O'Brien, University of California, Berkeley

E-mail: job@eecs.berkeley.edu

译者: 李鸣渤

审校: 刘永静

粒子系统、刚体模拟, 以及有关节物体模拟, 都已经成为商业游戏中比较普遍的东西了。这使得许多游戏中的物体可以回应游戏者, 做出程度不限的各种仿真行为。然而, 当物体发生弯曲、扭转、拉伸、挤压, 或者像图 3.8.1 中所示物体那样发生形状改变的时候, 就需要一些可变形物体模拟的形式。

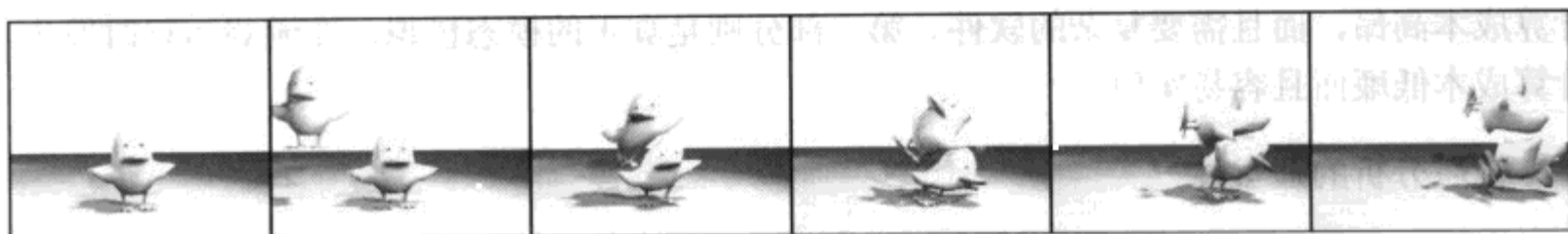


图 3.8.1 这些图片顺序显示了一些来自一对物体互相碰撞的动画的帧。每个物体都是一个合并了刚性和变形成分的混合模拟。变形成分使用了模态模拟

虽然简单且低分辨率的物体在游戏中很容易建模, 例如使用基本的弹簧质量块系统, 但是如果想要为更有趣味性、更复杂的物体使用过分简单的可变形模拟方法, 总是会出现一些由于缺乏真实性、计算成本过高, 以及(或者)缺乏稳定性所产生的困难。这些系统依靠时间数值积分来计算它们的行为, 而完善稳定的积分方案对于复杂性大的系统来说可能会十分昂贵。此外, 稳定问题会制约具有真实材质参数的庞大系统, 这些问题会造成时长非常小, 最终导致模拟时间慢得不切实际。

本文介绍了一种被称为模态分析(modal analysis)或模态模拟(modal simulation)的技术, 它可以以非常有效的方式模拟某些可变形的物体。支持这项技术的基本理论就是, 我们可以对一个已经存在的可变形模拟, 例如弹簧质量块模拟或者有限元模拟(finite element simulation)进行分析, 然后将其分为互不影响的不同数学组件, 我们称之为模式。由于这些模式之间互不影响, 我们可以单独分析它们每一个的行为, 当我们掌握了每一个模式的行为后, 那些行动不合要求的模式会被去掉。剩下的每一个模式都十分简单, 不需要进行时间数值积分, 只需简单的分析方法就可以计算

其行为。由于不使用时间数值积分，模态模拟本身并不会发生不稳定情况；由于不存在稳定性的问题，就可以使用大时长；同时由于即使在又大又复杂的模型中也只有少数的模式，因此每一步的计算成本都很低。举个例子，图 3.8.2 中所示的模型有 871 个顶点，但是当把它精简为一个只用 40 个模式的模态模拟时，模拟就可以很容易地实时运行在一台索尼 PS2 上，而且只使用一小部分 CPU。

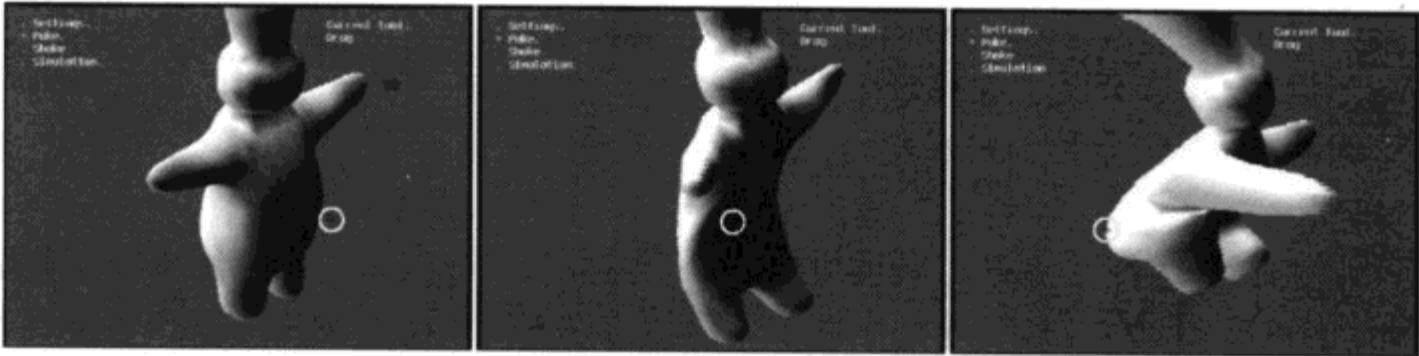


图 3.8.2 这些图片是运行在索尼 PS2 上的一个演示程序的屏幕截图。
加亮区的圆圈是光标，用户正在使用它拨弄和拖拉弹性的图形

使用模态模拟基本上是一个两阶段的过程。第一部分是模式分解，它在内容开发期间进行，计算成本高昂，而且需要复杂的软件。第二部分则是真正的模态模拟，在游戏运行时发生，计算成本低廉而且容易实现。

模态分析的适宜性

虽然模态模拟非常快速也非常成熟，但是这种技术有三个不同程度的基本局限性。第一个，也是最严重的局限是，在进行模式分解之前，原系统要先线性化。这使得可用于模态模拟的物体类型受到一些根本的限制。特别是在大量的弯曲或扭曲中含有非线性变形时，如果尝试用线性化的模态模拟来为其建模，就会使生成的物体发生明显的变形失真。图 3.8.3 就是一个变形的例子，一根直杆逐渐地被慢慢弯曲。在同一图中还可看到，小的形变看上去还能接受，但大的形变看起来又怪又夸张。由于这些失真的形变，因此模态模拟最适合于紧凑的物体，它们可以被挤压或者伸展，却不能大幅度地弯曲或扭曲。当然，对于有些模拟，如在类似卡通的游戏中，可能会需要用线性化的方法使物体形变得夸张些。

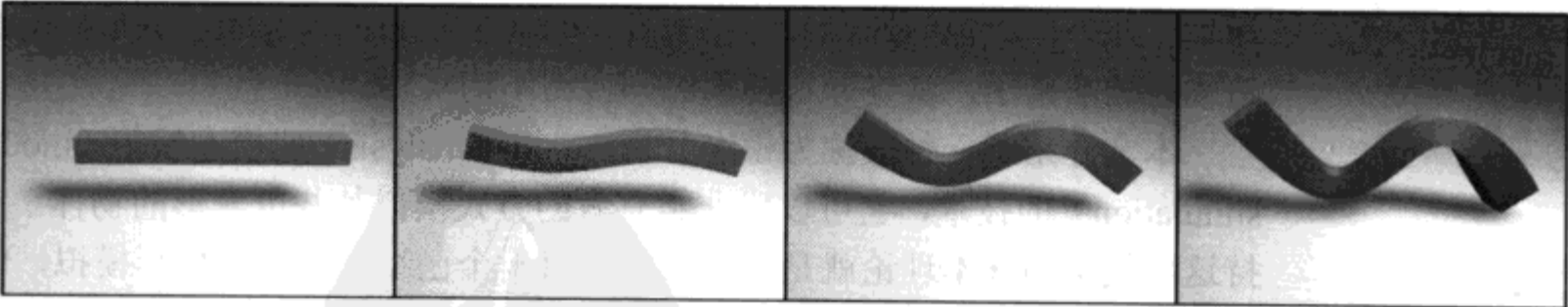


图 3.8.3 这些图形显示了在线性化中所介绍的那些形变。第一个图像显示了一个没有变形的棒。随后的一些图像则显示了变形逐渐增加的各种结果。一些适度的微小变形不会出现看得见的错误。然而，变形程度较大则会产生明显的形变。最后的图像显示了不仅形变明显，而且它的总长度也变化了

第二个局限性是，模式分解的计算需要大量的工作。我们呆会儿会向大家详细讲解分解

的计算,但最重要的是,它包含了对大的稀疏矩阵局部分解的计算。虽然这一过程的计算成本很高,但是分解可以在游戏开发期间预先被计算出来,所得出的模式作为物体的一部分储存起来。因此,只要游戏在不运行的时候不动态地产生新物体,这个局限性就不会产生任何问题。

模态分析的最后一个缺点就是,它依赖于一套相对先进的数学工具,因此对许多人而言,要理解其中的概念在刚开始的时候好像很难。值得高兴的是,实际上,虽然这个方法所含的数学知识很复杂,但是最终算法其实很简单。

3.8.1 模式分解

使用模态模拟的第一步就是需要得到一系列合适的用来描述一个已知物体的模式。这个过程要消耗不少计算,但是前面讨论过,这些计算只发生在游戏的开发阶段,而不是在游戏运行时。形象地说,我们是把应该在玩家机器上进行的计算工作,预先在开发者的机器上计算了。虽然在这里我们解释了分解的基本原理以便于你的理解,但是你可能还想使用一些第三方软件(建议如下),而不是完全由自己开发。

1. 最初的系统

我们假设用于被分解的可变形模拟的运动等式可以写成:

$$K(\mathbf{p}) + C(\dot{\mathbf{p}}) + M(\ddot{\mathbf{p}}) = \mathbf{f} \quad (3.8.1)$$

变量 \mathbf{p} 表示一个向量参数,它描述模拟的配置(例如,质量块弹簧模型或有限元模型的节点位置);一个上点表示对时间求导。 K 是位置的非线性函数,其返回值是弹力。例如,如果我们使用质量块弹簧系统, K 可以是计算作用于节点上的弹簧弹力的函数。 C 是速度(很可能也是位置)的非线性函数,其返回值是阻尼力。变量 \mathbf{f} 是诸如碰撞或用户输入而产生的外力向量。最后, M 是加速度的函数,其返回值是产生那些加速度所需的作用力;换句话说,它是对牛顿第二定律 $\mathbf{f} = m\mathbf{a}$ 的概括。

我们用这种非常笼统地形式介绍等式 3.8.1,这是因为我们并不关心是用哪种模拟方法来生成最初的一组等式。我们只用等式 3.8.1 来生成一组用来如何进行模拟的矩阵,然后接下来的所有操作都使用这些矩阵。这并不意味着结果不受原先模拟方法的影响:如果我们使用一个较差的方法,那都会出现在所得到的矩阵中,因而也会出现在模态模拟中。

2. 线性化

一旦我们使用等式 3.8.1 的形式对要模拟的系统进行描述后,下一步就是把系统线性化,使其能够被分解。这个线性化的步骤就是图 3.8.3 中所示的出现错误的地方。

将等式 3.8.1 进行线性化需要给等式中的每一项找出一个合适的线性近似值。对于第一项, $K(\mathbf{p})$, 我们会用到泰勒展开,然后丢弃所有二次或高于二次的项。这样就得到

$$K(\mathbf{p}) = K(\mathbf{p}_0 + \mathbf{d}) \approx K(\mathbf{p}_0) + K'(\mathbf{p}_0) \cdot \mathbf{d} \quad (3.8.2)$$

其中, \mathbf{p}_0 是我们展开的中心的位置, $\mathbf{d} = \mathbf{p} - \mathbf{p}_0$ 是从展开点到函数返回点的位移,而 K' 是被

称为 K 的雅可比行列式 (Jacobian) 矩阵的第一部分, 它由 $K_{ij} = \partial K_i / \partial p_j$ 得出。如果我们选择没有任何外力作用在初始位置 p_0 上, 那么 $K(p_0) = 0$, K 的线性化就变得很简单。

$$K(p) \approx K \cdot d \quad (3.8.3)$$

其中 K 被理解为与 p_0 的选择有关。矩阵 K 被称为系统的弹性矩阵, 它和大多数半隐性积分方案所需的雅可比行列式矩阵是一样的。

对于有限元或弹簧质量块系统, K 是通过把单个等式组合起来形成的, 因此一种决定 K 的简单方法就是为系统中的每个元素或者弹簧计算小元素的弹性矩阵, 然后用它们形成整个系统的全局矩阵。由于经常要用到这些矩阵, 大多数有限元课本中会为各种类型有用的元提供元弹性矩阵。

我们可以以类似的方式来决定 C 和 M 的线性化, 结果实现起来更加的容易。因为 M 只是对 $f=ma$ 的表达式, 它在多数对应于位置变量 p 的系统中已经是线性的了。大多有限元课本会把元质量矩阵与弹性矩阵一一并列出来, 要么也可以使用一个集总质量矩阵, 其中 M 是一个对角矩阵, 而在对角线上每项为相应节点的质量。由于 p_0 为常量, $\ddot{d} = \ddot{p}$, 我们可以写作 $M(\ddot{p}) \approx M \cdot \ddot{d}$ 。阻尼函数 C 可以是任意的, 但是有了一项被叫做 *Raleigh* 阻尼的技术, 阻尼矩阵 C 可以由两个系数 c_1 和 c_2 , 根据 $C = c_1 K + c_2 M$ 来决定。有了对 K 、 C 及 M 的线性化, 方程式 3.8.1 线性化了的版本就可以写成

$$K \cdot d + C \cdot \dot{d} + M \cdot \ddot{d} = f \quad (3.8.4)$$

或者, 如果我们用 *Raleigh* 阻尼代替 C 的定义:

$$K \cdot d + (c_1 K + c_2 M) \cdot \dot{d} + M \cdot \ddot{d} = f \quad (3.8.5)$$

3. 特征值分解

我们把等式 3.8.5 拆分为其模式的这一步是通过解决一个被视为笼统的特征值问题来完成的。出于篇幅的考虑, 在这里我们只对结果作总结, 有关的更详细的解释, 读者可以在 [O'Brien02]、[Maia98] 或者大部分探讨有关有限元方法的文章中找到。

矩阵 K 是相对称的, 而 M 不但对称, 而且正定, 因此我们可以找到一个可逆矩阵 W , 以及对角矩阵 A , 如

$$K \cdot W = M \cdot W \cdot A \quad (3.8.6)$$

矩阵 W 的列是系统的总特征向量, 而 A 中相对应的是其特征值。如果我们事先用 W^T 与等式 3.8.5 相乘, 那么就可以使用以下替换方法把 d 和 f 分别替换为 z 和 g :

$$z = W^{-1} \cdot d \quad d = W \cdot z \quad g = W^T \cdot f \quad f = W^{-T} \cdot g \quad (3.8.7)$$

然后对结果进行简化, 我们就能得到以下对角等式:

$$A \cdot z + (c_1 A + c_2 I) \cdot \dot{z} + \ddot{z} = g \quad (3.8.8)$$

我们把 d 看作在原先形式中表示的值, 或者说空间坐标, 同时把 z 看作在模式坐标中表示的值。

这个转化背后的概念与 3D 坐标系统间转换的思维是相同的: 任何基础都没有更改, 我

们只是工作在一个不同的坐标系下。然而，当使用的坐标系统合适的时候，一些图像问题就变得容易了许多；同样，在模式坐标，而不是在空间坐标中进行模拟也会有着本质上的受益。因为等式 3.8.8 是对角的，所以如果我们有了 W 和 A ，解决问题的时候用对角方程式先把空间坐标转化为模态坐标，然后把结果转化回空间坐标，与在空间坐标中直接解决同样的问题相比，效率就可以大大提高。对傅立叶变换（FFT）很有经验的读者应该会觉得这一思路似曾相识，因为在某种意义上，模式转化就是一种笼统的傅立叶变换。当然， W 和 A 的计算包含了大量工作（即寻找特征值系统），但是由于所有矩阵都是恒定的，这项工作只需预先进行一次就可以了。

4. 解析解法

虽然使用对角矩阵有许多长处，但是产生这种分解方法真正优点的原因是，等式的一个对角系统实际上是一组分离的等式，它们互相之间不影响。换句话说，等式 3.8.8 的每一行都独立于其他行，而且它们每一个都能写成一个二阶微分等式：

$$\lambda_i z_i + (c_1 \lambda_i + c_2) \dot{z}_i + \ddot{z}_i = g_i \tag{3.8.9}$$

与这种等式相似的解法也很出名：

$$z_i = a_1 e^{i\omega_i^+ t} + a_2 e^{i\omega_i^- t} \tag{3.8.10}$$

其中 t 是时间， a_1 与 a_2 为常量，它们会在初始化 z_i 后得出，而 ω 是通过以下方程得到：

$$\omega_i^\pm = \frac{-(c_1 \lambda_i + c_2) \pm \sqrt{(c_1 \lambda_i + c_2)^2 - 4 \lambda_i}}{2} \tag{3.8.11}$$

我们应该避免出现根号下的数值为零时的特殊情况，解决的方法就是对其值进行检测，看它是否接近零，如果是，就把它替换为一些具有相同符号的最小值（如 10^{-4} ）。能够通过对比等式 3.8.10 进行微分来求得系统的速度是非常有用的，它是这样的：

$$\dot{z}_i = a_1 \omega_i^+ e^{i\omega_i^+ t} + a_2 \omega_i^- e^{i\omega_i^- t} \tag{3.8.12}$$

3.8.2 模式的理解和丢弃

每一个模式对应 W 的一列，同时，就如我们区分一个 z_i ，物体的形状可以通过为相应的 W 列给出的物体增加一个替代场而发生改变。如图 3.8.4 所示， W 的列为基本形状，任何配置的物体都可以由这些基本形状组合而成。

我们也可以通过等式 3.8.10 和 3.8.11 看看每个模式表现如何。如果等式 3.8.11 中根号下的数为负，那么 ω^+ 和 ω^- 则是复数共轭，而方程式 3.8.10 则会表现为衰减的正弦曲线。这种情况被看作不完全衰减。 ω 的虚数部分决定了根号下每秒的模态频率，而实数部分（负的）则决定这一模态的震动衰减得速度。如果根号下的数为正，那么 ω^+ 和 ω^- 为两个绝对值相同的正负数。这种情况被看作强阻尼，而结果表现为指数衰减至零。

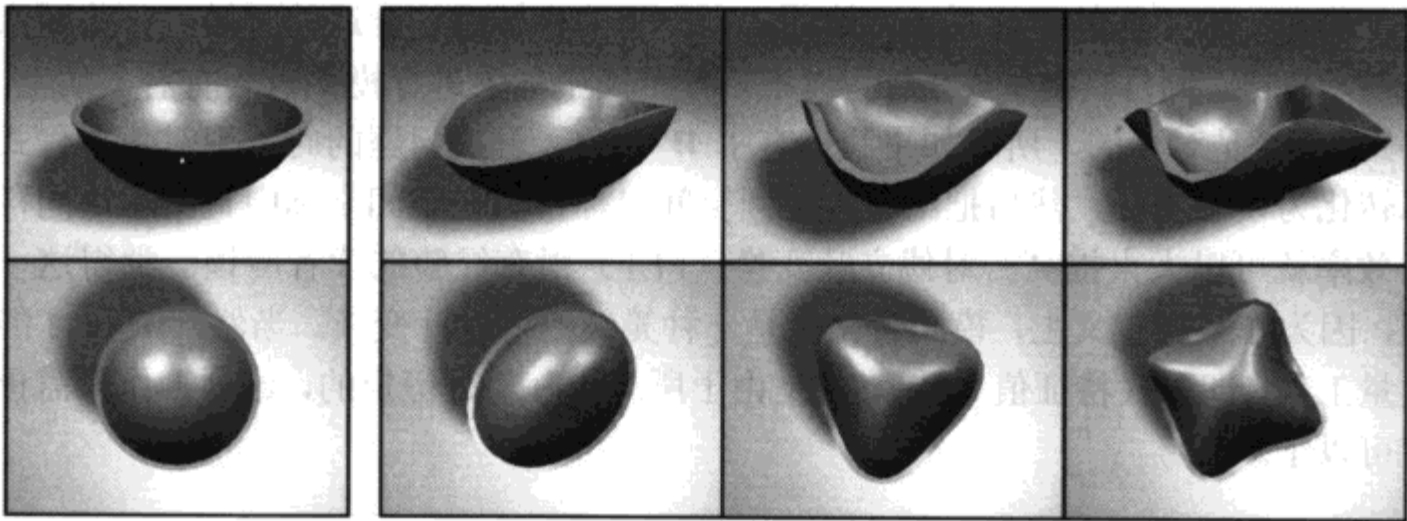


图 3.8.4 这两行分别显示了具有 3 种碗的优先（按照特征值排序）摇摆模式的碗的侧视图和俯视图。

例子中所选择的模式是具有不同的特征值的优先的 3 种非刚性模式，
它们通过横向推动碗的边缘产生

一旦我们找出每个模式的行为特性后，就可以通过考虑是否需要每个模式而得到更多有用的东西。特别是，我们可以把那些在我们模拟现象中发生不明显的模式除去。如果与某一特定模式相联的特征值 λ_i 很大，那么用来造成这个模式中有明显位移的力也会很大。在一个已知环境中，我们知道力的大小中会有一个上限，而可察觉运动的幅度中会有一个下限。举例说明，如果模拟一个室内环境，我们知道不会出现超过 60 000N 的力（即损坏大型卡车的力），我们也不能观测到小于 0.1mm 的位移。因此，假如 w 是 W 的一列，而 λ 是相应的特征值，如果 $\|w\| / \lambda < \text{minResolution} / \text{maxForce}$ ，那么我们就可以把这个效果不明显的模式从模拟中忽略掉。另外，不完全衰减模式中震动大与显示的刷新率的一半时会引起难看的锯齿现象，所以删除他们实际上会优化模拟的结果。

对于大多数物体，几乎所有的模式都是要么有足够的硬度，要么有足够高的频率，才可以去处理。一个典型的情况就是有几个个节点的物体有大约 50 个模式需要保留。此外，必须保留的模式数量几乎不依赖于模型的精细度。因此，即使是非常复杂的模拟模式也可以精减成为数甚少的模式，这样的精减可以在模拟中节约巨大的计算量。

$\lambda = 0$ 的模式也是可能存在的。这样的模式是典型的物体刚体模式，它们与物体所具有的无论怎样的移动或转动的自由度相符合。单个自由浮动 3D 物体有 6 个对应 3 个移动、3 个转动自由度的零模态。由于它们可以在别处被处理，因此刚体模态也可以被去除（参见混合模拟一节）。

3.8.3 模态模拟

对等式 3.8.10 中给出的每个模式的解决方案决定了那种模式随时间的表现。如果我们已知 a_1 和 a_2 适当的值，那么要得出模拟的行为只需在每个保留的模式中求出等式 3.8.10 的值，以得出 z ，然后用等式 3.8.7 来重新映射出原有的空间坐标。

1. 更新模式

我们假设在某个时间 t ，在方程式 3.8.10 中代入 a_1 和 a_2 的值可以返回每个模态的当前值

z_i 。如果我们想要计算每个模式新的 z_i 值，最明显的方法就是更新 $t=t+\Delta t$ ，然后重新计算方程式 3.8.10。但是这种方法并不理想，有两个原因：首先，指数计算相对昂贵；其次，当我们施加外力时，方法会变得相当复杂。还记得 $e^{a+b}=e^a e^b$ 吗？我们在把它应用到想要的更新中，就得到：

$$a_1 e^{(t+\Delta t)\omega_i^+} + a_2 e^{(t+\Delta t)\omega_i^-} = a_1 e^{t\omega_i^+} e^{(\Delta t)\omega_i^+} + a_2 e^{t\omega_i^-} e^{(\Delta t)\omega_i^-} \quad (3.8.13)$$

也就是说，我们可以简单地通过把它们每一项乘以复数来更新它们。如果 Δt 和通常情况一样，在每帧是常量，那么刚才作乘数的值也是常数，而且可以预先计算。

要利用等式 3.8.13，我们可以为每个模式存储 4 个值。前两个 ϕ_i^- 和 ϕ_i^+ ，存储着 $a_1 e^{t\omega_i^+}$ 和 $a_2 e^{t\omega_i^-}$ 的当前值。后两个 ϕ_i^+ 和 ϕ_i^- ，存储着 $e^{(\Delta t)\omega_i^+}$ 和 $e^{(\Delta t)\omega_i^-}$ 预先计算的值，在经过时间 Δt 的时候，只需要对每一模式做一对复数乘法 $\phi_i^+ = \phi_i^+ \phi_i^+$ 和 $\phi_i^- = \phi_i^- \phi_i^-$ 就可以更新这些值。每一次需要 z_i 的值的时候，我们只是简单地对 ϕ_i^- 和 ϕ_i^+ 求和。速度可以分别由 ϕ_i^- 和 ϕ_i^+ 同 $\dot{\phi}_i^+$ 和 $\dot{\phi}_i^-$ 相乘，然后求和得出结果。

2. 初始条件及外力

如果一个模拟以零速度并且 $p=p_0$ 开始，那么 ϕ_i^- 和 ϕ_i^+ 的初始值就简单地为零。然而，这对于从其他一些非静止状态开始的模拟很有用。也就是说，我们可能会根据模拟开始的地方设置初始值 p 和 \dot{p} 。 z 和 \dot{z} 的初始值计算也很直观。

$$z = W^{-1} \cdot (p - p_0) \quad \dot{z} = W^{-1} \cdot \dot{p} \quad (3.8.14)$$

以下等式给出了每一种模式 ϕ_i^- 和 ϕ_i^+ 的合适值：

$$\phi_i^\pm = \frac{z_i}{2} \pm \frac{(c_1 \lambda_i + c_2) z_i + 2 \dot{z}_i}{\sqrt{(c_1 \lambda_i + c_2)^2 - 4 \lambda_i}} \quad (3.8.15)$$

要在系统中施加一些外力 f ，我们首先假设在一些时间区间中力为常量，算出所得的冲量 $\Delta t f$ ，然后用 $\Delta t g = W^T \cdot (\Delta t f)$ 把冲量转化为模式坐标。冲量会导致模式速度发生变化。由于整个系统是线性的，我们可以计算 ϕ_i^- 和 ϕ_i^+ 上的变化，方法就是使用 z_i 为零的等式 3.8.15 以及设置 \dot{z} 和 $\Delta \dot{z}_i$ ，然后简单地把那些值加在已知的 ϕ_i^- 和 ϕ_i^+ 值上。

3. 混合模拟

前面，我们把物体的刚体模式搁在一边没有处理，因为模态模拟不是模拟刚体行为的最佳工具；标准的刚体模拟方法使用起来更好。如果我们要模拟一个即会移动又能变形的物体，可以选用最适合于物体运动的某个方面的模拟工具来建模。因为我们知道刚体模态本质上是从变形模态分离而来的，我们知道用这种方法把模拟分开来可使工作进行顺利。

这个被称为混合模拟的基本思路，最早在 [Terzopoulos88] 中提出。一个标准的刚体模拟是为有合适的质量、转动惯量物体而建立的。然后，可变形模拟（例如模态模拟）嵌入到刚体局部坐标框架中。任何加在物体上的外力应该同时作用在刚体和模态模拟中。

3.8.4 总结

前一节解释了建立基本模态模拟所牵涉的一些细节。下面这个大纲对如何把所有要素结合起来作了总结。

I. 在制作的时候，预先计算物体的模态描述。

- (1) 为想要建模的物体建立弹簧质量块模拟或有限元模型 (FEM) 模拟。
- (2) 以模型为基础，决定已知的其他参数的系统矩阵 (K 、 C 及 M)。
- (3) 计算相应的模态及特征值 (W 、 W^{-1} 及 Λ)。
- (4) 去掉不需要的模态，算出其余模态的 ω_i^{\pm} 。
- (5) 如果帧速率是恒定的，则预先计算 ϕ_i^{+} 和 ϕ_i^{-} 的值。

II. 玩游戏的时候，启动模拟。

- (1) 确定模拟的初始状态 (如， p 和 \dot{p} 的初始值)。
- (2) 用方程式 3.8.14 及 3.8.15 计算每个模式的初始 ϕ_i^{-} 和 ϕ_i^{+} 。
- (3) 在每帧动画中：
 - (a) 算出 f 及所有作用在物体上的外力；
 - (b) 计算 $\Delta \dot{z} = W^T \cdot (\Delta t f)$ ，并用方程式求出每个 ϕ_i^{-} 和 ϕ_i^{+} 的增量；
 - (c) 通过设置 $\phi_i^{\pm} = \phi_i^{\pm} \phi_i^{\pm}$ 及时把系统向前推进；
 - (d) 通过 $z_i = \phi_i^{+} + \phi_i^{-}$ 计算 z 的新值；
 - (e) 更新 $p = p_0 + W \cdot z$ ；
 - (f) 用 p 中的坐标显示物体。

在实际运用中，模拟运行时间这一部分中的计算成本最昂贵的部分就是由 z 重新构造 p 这一步。这一操作实际上是计算基本形状的加权融合。但是很明显，这比原来的模拟成本要低许多。要记住，只使用与保留模式相对应的 W 列和 W^{-1} 行。这一操作也可以使用硬件加速来进行 (参阅 [James02])。

3.8.5 结论

这篇文章提供的代码使用了前面一节介绍的大纲中的实际模拟部分。同时它还实现了混合模拟、限制器及碰撞。有关用于限制器及碰撞的具体方法，读者可以参阅 [Hauser03]。

另外还有一些包含模式分解的模型案例。假如你已经有了用于弹簧质量块模拟或有限元模型 (FEM) 模拟的现成代码，那么首要具备的工具就是一个好的程序，用于计算稀疏对称矩阵的特征分解值。对小系统而言，选择有不少，包括 Matlab、Numerical Recipes [Press02] 或 LAPACK。对大系统而言，我们应该使用专为大的稀疏矩阵设计的软件，如 ARPACK 或 TRLAN。有了这些软件，就可以确定你感兴趣特征值的范围，那么就可以避免对你准备丢弃的模式进行计算。[Bai99] 是一个不错的有关特征分解值计算方法的综合参考资料。另外，许多有限元软件包，如 Nastran、ABAQUS 或 ANSYS 也含有计算模式分解的工具。

有很多优秀的教材讨论模式分析。[Maia98] 中有关模式分析的文章，除了讨论实际应用外，还详细深入地介绍了模式分析的数学及物理理论。有关有限元方法的著作 (如

[Cook89]) 也有专门针对模式分析的章节, 当然还有对有限元方法的重要讨论。最后, [Pentland89]、[Stam97]、[O'Brien02]、[James02]、[Hauser03] 这几篇文章都已经发表了, 它们讨论了如何在交互应用中使用模态模拟。

3.8.6 参考文献

[Bai99] Bai, Z., et al., "Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide," SIAM, Philadelphia, 1999.

[Cook89] Cook, R. D., D. S. Malkus, and M. E. Plesha, *Concepts and Applications of Finite Element Analysis, Third Edition*, John Wiley & Sons, New York, 1989.

[Hauser03] Hauser, K.K., C. Shen, and J. F. O'Brien, "Interactive Deformation Using Modal Analysis with Constraints," *Graphics Interface 2003*, June 2003, pp. 247–256.

[James02] James, D.L., and D. K. Pai, "DyRT: Dynamic Response Textures for Real Time Deformation Simulation with Graphics Hardware," *SIGGRAPH 2002*, August 2002, pp. 582–585.

[Maia98] Maia, N., and J. Silva, *Theoretical and Experimental Modal Analysis*, Research Studies Press, Hertfordshire, England, 1998.

[O'Brien02] O'Brien, J. F., C. Shen, and C. M. Gatchalian, "Synthesizing Sounds from Rigid-Body Simulations," *SIGGRAPH 2002 Symposium on Computer Animation*, July 2002, pp. 175–181.

[Pentland89] Pentland, A., and J. Williams, "Good Vibrations: Modal Dynamics for Graphics and Animation," *SIGGRAPH 89*, July 1989, pp. 215–222.

[Press02] Press, W.H., et al., *Numerical Recipes in C++, Second Edition*, Cambridge University Press, 2002.

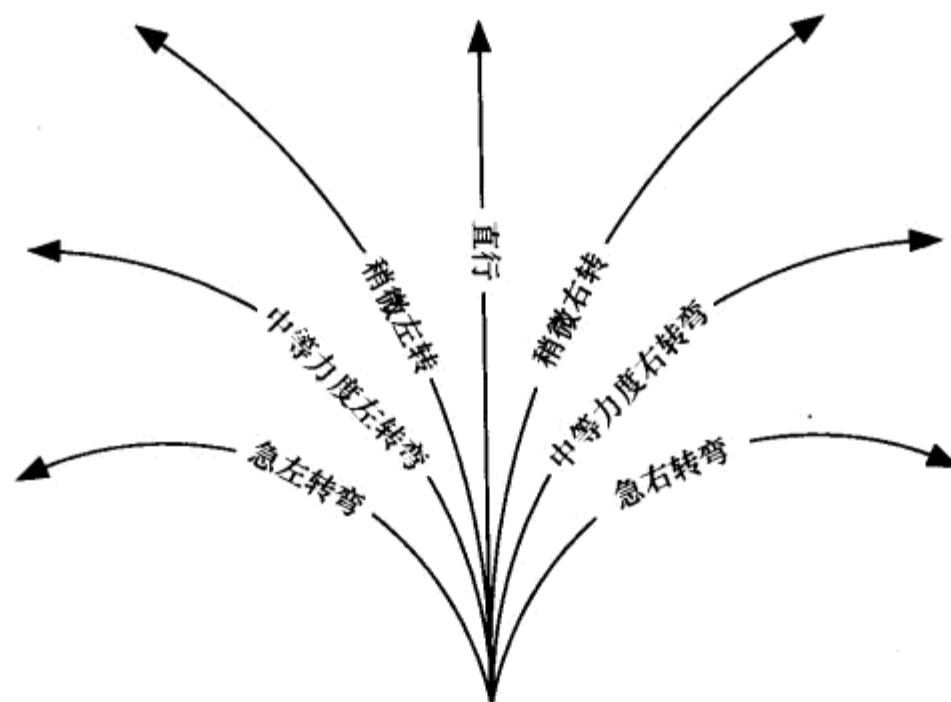
[Stam97] Stam, J., "Stochastic Dynamics: Simulating The Effects of Turbulence on Flexible Structures," *Computer Graphics Forum*, 16(3): August 1997, pp. 159–164.

[Terzopoulos88] Terzopoulos, D., and K. Fleischer, "Deformable Models," *The Visual Computer*, 4(6):1988, pp. 306–331.



人工智能

可能的转弯弧度



简介

作者: Paul Tozour, Retro Studios / Nintendo

E-mail: ptozour@austin.rr.com

译者: 刘永静

审校: 沙鹰

最近,我在几所大学里进行了题为“游戏 AI 入门”的演讲。这个演讲大约有两个小时的内容,在这里我就不将它翻出来再讲一遍了。演讲的内容归结于一点:游戏中的人工智能(AI)不是一个单一的问题,而是一个巨大的问题集。这问题集里的问题不仅数量多,而且在每个游戏中还都不相同。AI 完全依赖于游戏设计,几乎游戏中的每个新设计都会带来一定 AI 方面的问题。要找 50 款完全不同的游戏或许不很容易,但如果按照 AI 的不同来区分游戏的不同,那么要找出 50 款完全不同的游戏就得出奇地容易了。

本章较好地反映了游戏 AI 的多样性。John Olsen 介绍了吸引子(attractor)和排斥子(repulsor)的概念。Karén Pivazyan 解释了一些方法,帮助我们使用动态编程来处理随机性。Borut Pfiefer 则提出了一些途径,用于通过程序调节戏剧张力(dramatic tension)的级别来系统地平衡游戏的难度。Jonathan Stone 贡献了一篇关于运镜技巧,即第三人称视角摄像镜头的运动规则(third-person camera navigation)的详尽介绍。

布尔逻辑、离散的“若……就(if-then)”规则,以及确定性的有限状态机作为游戏 AI 的核心,已经有相当长一段时间了。然而,随着这个行业的不断成熟和我们对游戏 AI 开发标准的不断提高,将越来越需要找到一些方法,让我们的 AI 系统更接近于“软性计算”——也就是说,我们需要一种能够处理非确定性、部分真值表和模糊的系统。它也应能够分析大量规则的输出,并把结果联结起来以形成连续的输出,而不只是简单地测试一个 if-then 规则序列。

为了这个目标,一些 AI 开发人员运用了各种技术,比如模糊逻辑(fuzzy logic)、贝叶斯置信网络(bayesian belief network)、单纯贝叶斯分类器(naïve bayesian classifier),以及 Dempster-Shafer 理论等,让我们的游戏 AI 更接近于“软性计算”。在很多情况下,只要我们能够完全理解面临的问题,并且巧妙地“量体裁衣”设计出有效的方案,我们都可以使用较简单的方法而取得较好的结果。从这个意义上讲,我觉得 John Hancock 写的关于分布式推理和基于工具的决策架构的文章非常有用。

每次我在作“游戏 AI 入门”的演讲时,总有人问我同样一个问题:

现在我们有了硬件图形加速器，但为什么没有人设计硬件 AI 加速器呢？从硬件图形加速器的巨大成功可知，AI 领域显然也可以由某种硬件加速器而获益。

对此我的回答总是：“不”。我认为这是一个软件问题，增加计算能力并没有太多的帮助。而且就游戏 AI 而言，它也是一个有关开发成本和行业成熟度的问题。再者说了，随便找来 50 个游戏，其 AI 可能全都不尽相同——因此实在很难想象，我们可以使用某种特殊处理器，从而瞬间提升其中哪怕只有十分之一的游戏 AI。

无疑，问题是有待解决的。我说“不”不是因为我的确这样认为。我只是想要让人理解游戏 AI 始终需要灵活的通用硬件。技术在不断地进步，在游戏 AI 领域，我们也必将找到更好的方法来利用这些日益增长的计算能力。可编程的硬件确实为游戏 AI 的发展开拓了无尽的可能性，Thomas Rolfes 为那些对探索当前技术发展水平感兴趣，也对未来隐藏的机遇感兴趣的人们，提供了一个极好的起点。



4.1 第三人称视角摄像镜头的运动规则

作者: Jonathan Stone, Double Fine Productions

E-mail: jon@doublefine.com

译者: 沙鹰

审校: 李劲松

在探索三维游戏世界时可用的众多视角中, 第三人称透视视角是最为动人的。通过对玩家角色运用一个外部视角, 比起仅仅使用第一人称视角, 我们能够在玩家和环境之间创造更为详细的交互行为, 同时传达一种更强烈的玩家的本体感。

对于复杂场景来说, 为了使第三人称视角的摄像镜头(以下简称第三人称 camera 或 camera) 与世界平稳地交互, 我们必须解决一些技术难题。虽然简单地通过限制 camera 在固定位置或预定义的路径上就可以解决问题, 但是这些方案限制了用户漫游场景的自由度。为了实现对复杂场景的自由漫游, 有必要设计一个动态的、用户可控的 camera。

在本文中, 我们将略述构造一个动态第三人称 camera 系统的基本步骤, 并描述 camera 漫游场景时发生的一些有关场景边界体设定(scene-bounding) 和遮断(occlusion) 的棘手问题的解决方案。

4.1.1 Camera 定位及运动

我们的第三人称 camera 系统的基础是追尾 camera(chase camera), 追尾视角的设计一直在游戏中得到利用, 从经典 platform 游戏如 *Zelda 64*、《雷曼 2》(*Rayman 2*) 直到 *Jak and Daxter* 和《分裂细胞》(*Splinter Cell*)。当玩家在游戏世界里蹦蹦跳跳的时候, 追尾 camera 通常就在玩家的后上方跟随, 但对用户输入也是有反应的, 允许玩家检查邻近的细节或通过调节 camera 角度来判断距离。

首先, 在玩家角色身上选择一个目标位置, 将其用作 camera 旋转的 look-at 点, 同时也是 camera 位置的一个参考点。具有代表性地, 这位置可能会在玩家身体的边界柱形体上靠近顶部的某处, 也可能在柱形体正上方某处。

然后, 我们需要在 camera 和此位置间定义一个“理想的”空间关系。例如, 我们要求 camera 和目标之间的距离保持 ρ 米, 并从上方与水平面保持呈 φ 度夹角。

1. 球面坐标

一种有效的描述此类关系的表示法是球面坐标系。在球面坐标系中，3D 空间中的每个位置都被描述为与目标点之间的距离，以及相对该点的旋转角度。一个球面坐标矢量有三个分量：和原点之间的距离 ρ 、逆时针绕 z 轴的旋转角度 θ （即经度）、与 xy 平面相夹的有符号旋转角度 ϕ （即纬度）。纬度分量在不同的应用中有不同的表示法，但在本文中我们令 ϕ 等于从原点望去的视线与 xy 平面之间所成夹角的带正负号的角度值。高于 xy 平面的位置的纬度处于 0 度~90 度之间，而低于 xy 平面的位置的纬度处于 0 度~-90 度之间。

在球面坐标系中，我们可将 camera 与目标之间的理想偏移量描述为矢量 (ρ, θ, ϕ) ，并逐帧将其转换成笛卡尔世界坐标系中的一个位置。从球面坐标系坐标换算为笛卡尔坐标系坐标的公式是：

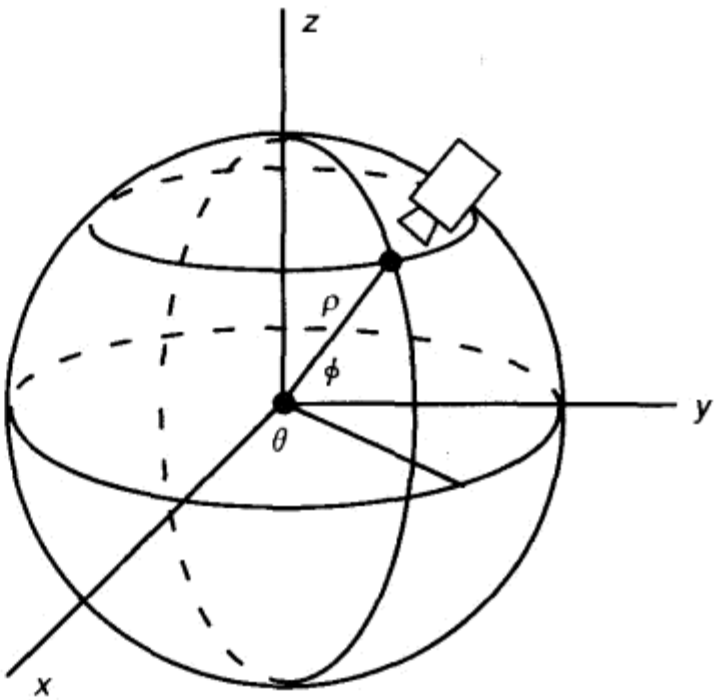


图 4.1.1 追尾视角在球面坐标系中的表示。[注意此图有错]

$$x = \rho \cos \theta \sin \left(\phi + \frac{\pi}{2} \right) \tag{4.1.1}$$

$$y = \rho \sin \theta \sin \left(\phi + \frac{\pi}{2} \right) \tag{4.1.2}$$

$$z = \rho \cos \left(\phi + \frac{\pi}{2} \right) \tag{4.1.3}$$

从笛卡尔坐标系坐标换算为球面坐标系坐标，可以用下列公式：

$$\rho = \sqrt{x^2 + y^2 + z^2} \tag{4.1.4}$$

$$\theta = \tan^{-1} \left(\frac{y}{x} \right) \tag{4.1.5}$$

$$\phi = \cos^{-1} \left(\frac{z}{\rho} \right) - \frac{\pi}{2} \tag{4.1.6}$$

2. 弹簧系统

当我们定义了 camera 在世界中的理想位置后，接下来就需要一个插值方法。由于系统是全动态的，也就是说目标和 camera 本身的位置每帧都在改变，我们不能简单地按固定长度的时间来插值。实际上，我们需要一个加速度系统，每帧影响 camera 的速度，从而即使在目标本身处于移动的时候，仍得以维持连续性。阻尼弦系统就是一个这样的模型，施加弦加速度于在物体离开静止位置位移的相反方向上，同时施加阻尼加速度在速度的反方向上

[Treglia00]。等式可以写作：

$$F=ma=-k_sx-k_dv \quad (4.1.7)$$

式中 x 代表位移， v 代表物体的速度， a 代表合加速度。 k_s 是弹性系数， k_d 则是阻尼系数，分别用于控制加速度的弹性及阻尼分量。对 camera 系统而言， x 代表 camera 在世界坐标系中离开理想位置的距离，而最终合成的加速度 a 将驱动 camera 平稳地靠近其理想位置。

为弹簧系统选择弹性系数和阻尼系数的时候，应当留意系统的阻尼比率，即：

$$\xi = \frac{k_d}{2\sqrt{k_s}}$$

当 ξ 等于 1 时，该弹簧系统称为是临界阻尼的，系统将会在对应 k_s 值的尽可能短的时间内回归静止位置。若 ξ 小于 1，该系统称为是弱阻尼（或欠阻尼）的，将在到达静止位置后继续振荡。若 ξ 大于 1，该系统是过阻尼的，达到平衡状态将需要比必须更长的时间。在 camera 系统中，我们一般会使用临界阻尼弦，因为这样能得到最经济的 camera 运动。

3. 更新 camera

现在我们要设计一个逐帧更新 camera 的函数，使其朝玩家的方向渐渐恢复理想的偏移量。像这样定义我们的简化的追尾 camera 类：

```
class ChaseCamera
{
    Vec3 m_vPosition;           // camera 位置
    Vec3 m_vVelocity;           // camera 速度
    Vec3 m_vTargetPos;          // 目标位置
    Vec3 m_vIdealSpherical;      // 理想球坐标
    Mat4 m_mView;               // 视矩阵
};
```

定义好类之后，我们这样编写 camera 的更新函数：

```
void ChaseCamera::Update(float fTime)
{
    // 根据 camera 的当前位置以及目标的方位来更新理想的方位角
    m_vIdealSpherical.y = atan2f(
        m_vPosition.y - m_vTargetPos.y,
        m_vPosition.x - m_vTargetPos.x);

    // 计算 camera 在世界坐标中的理想位置
    Vec3 vIdealPos = m_vTargetPos +
        SphericalToCartesian(m_vIdealSpherical);

    // 计算朝向理想位置的弦加速度
    Vec3 vDisplace = m_vPosition - vIdealPos;
    Vec3 vSpringAccel = (-m_fSpringK * vDisplace) -
        (m_fDampingK * m_vVelocity);

    // 用欧拉积分来更新 camera 的速度和位置
```

```

    m_vVelocity += vSpringAccel * fTime;
    m_vPosition += m_vVelocity * fTime;

    // 构造视矩阵
    m_mView = MatrixLookAt(m_vPosition, vTargetPos,
        GetWorldUpVector());
}

```

本函数中第一个调用的作用是，逐帧更新理想的经度角，使追尾 camera 绕目标自动地旋转。这是一种“惰性地”旋转，但足以使追尾 camera 在玩家绕圈行走时保持稳定，比如在 *Rayman 2* 那样的 platform 游戏中。若不进行该函数调用，camera 就会以平移的方式运动，无论玩家走到何处经度角都固定为某个值。这样的平移行为在以 *Splinter Cell* 为代表的一些需要实现精确瞄准的第三人称视角游戏中得到了良好的运用。

为了允许玩家从有利于解决谜题的各种角度来观察场景，也为了允许玩家实时地控制观看动作的视角，一定程度的用户控制是必需的。我们可以在上述 update 函数的头部添加下列代码，以赋予用户对 camera 的经度角的控制。

```

// 从用户的输入及速率衰减中计算出方位角的加速度
float fAzimuthAccel =
    (GetUserInput(kINPUT_AZIMUTH) * m_fAzDriveK) -
    (m_fAzimuthVel * m_fAzDampingK);

// 更新 camera 的方位角速度
m_fAzimuthVel += fAzimuthAccel * fTime;

// 作用于 camera 的位置上
Vec3 vCurSpherical = CartesianToSpherical(
    m_vPosition - m_vTargetPos);
vCurSpherical.y = NormalizeAngle(vCurSpherical.y +
    m_fAzimuthVel * fTime);
m_vPosition = SphericalToCartesian(vCurSpherical);

```

这实现了一个受驱策的、有阻尼的系统，用户的输入提供了 camera 经向加速度，阻尼效果则使 camera 的经向速度缓缓降低为零。GetUserInput()函数返回值的范围在-1~1 之间，反映输入设备的状态，例如模拟量游戏机操纵杆 (analog console joystick) 当前所处的左右位置。然后游戏策划可利用常数 m_fAzDriveK 以及 m_fAzDampingK 来决定这样控制的速度和反应敏捷程度。在本系统中，camera 的最大径向速度是当操纵杆被扳到最左边或最右边时达到的 m_fAzDriveK / m_fAzDampingK，单位是弧度每单位时间。

如果同时希望用户控制 camera 的纬度，可以添加一个系统来解释这额外的纬向控制。这对于仰望天空或俯视陡峭的峡谷谷底是很有用的。

4.1.2 Camera 与场景边界

我们已略述了一个能在全开放的场景中持续追随玩家的追尾视角 camera 的实现。但是，真实场景中会存在诸如墙壁和天花板那样的必须阻止 camera 穿越的边界。如果让 camera 离

开场景,可能会最终导致从墙壁后渲染游戏场景,这就打破了这个世界是一个“实在的世界”的感觉。为了将 camera 限制在可正确渲染的世界内,首先要定义何为场景的内部(interior)和外部(exterior)。

1. 碰撞几何体

如果我们用来渲染的场景模型是毫无限制的各类多边形的堆砌,那么我们需要单独设置一批碰撞几何体的类型来划分场景的边界。碰撞几何体通常具有比实际被渲染的世界少得多的面,并且为了优化“某点在场景中与否”此类测试而设计成滴水不漏的(或二阶流形, *2-manifold*)。对面为三角形的网格多面体(mesh)来说,滴水不漏(水密性, watertightness)意味着每个三角形在其三条边的每一条上均仅与一个其余的面相邻,且此网格不包含 T 形连接(T-junction)或重叠的多边形。

有了这些水密的碰撞几何体,我们可以将场景的内部定义为一些满足条件的点的集合,条件为从该点朝任何方向发出的射线均首先相交于碰撞几何体上某个朝向该点的面(也就是说,一个面法向与射线方向相反的多边形表面)。类似地,我们将场景的外部定义为一些从该点朝任何方向发出的射线或不与任意多边形相交,或与几何体上某个法向背向该点的面相交。从而,我们能够快速地判断世界上的任何给定位置是在场景的内部还是外部,方法是从任意方向发出一条射线,与最先相交到的多边形表面的面法向作比较。

2. 动态球

迄今为止,我们一直将 camera 当作在游戏世界中移动的单个点来处理。但是为了处理场景边界体设定,最好是赋予 camera 一定的体积。若我们为方便相交处理而将 camera 表示成半径为 r 的球体,就能保证处在球心的视点(viewpoint)能和场景边界保持最小距离 r 。若 r 的值充分大,我们就能避免 camera 视锥(frustum)的近表面(near plane)在场景的边缘与渲染后的多边形发生裁剪。

为了计算移动中的 camera 球与碰撞几何体的相交,我们需要一种球体和三角形相交的动态测试方法,方法是让球体沿一条线段运动,判断球体是否在该段运动路径上的某一点与任一三角形相交。若存在相交,测试即返回这条线段上某个最远的球体刚好掠过而不发生碰撞的位置。球体沿线段运动时所占据空间的总和称为线扫掠球体(line-swept-sphere) [Moller02]。

场景的碰撞几何体保存在一个分级的场景图(hierarchical scene graph)中,这向我们提供了对游戏世界中特定区域的多边形的有效访问途经,使我们将碰撞功能扩充为一个动态的“球和场景”之间的测试。此项测试是我们 camera 碰撞算法的关键成分。其机能类似射线投射,但不同之处在于它允许我们赋予射线以半径 r 为“宽度”,见图 4.1.2。

通过使用这些相交测试,下面我们将介绍两种 camera 场景边界体设定的算法,分别称作虚拟的和物理的碰撞模型。

3. 虚拟 camera 碰撞

第一种方法称为虚拟碰撞模型,是实现起来最简单的方法之一。采用这种方法时, camera 的碰撞是这样处理的。从目标朝 camera 发出一条假想的光线。每一帧,将一个球体从目标位

置开始向 camera 的位置扫掠过来。如果与场景发生碰撞，则将碰撞发生的位置作为 camera 的新位置。若不发生碰撞，则不改变 camera 的位置（见图 4.1.3）。

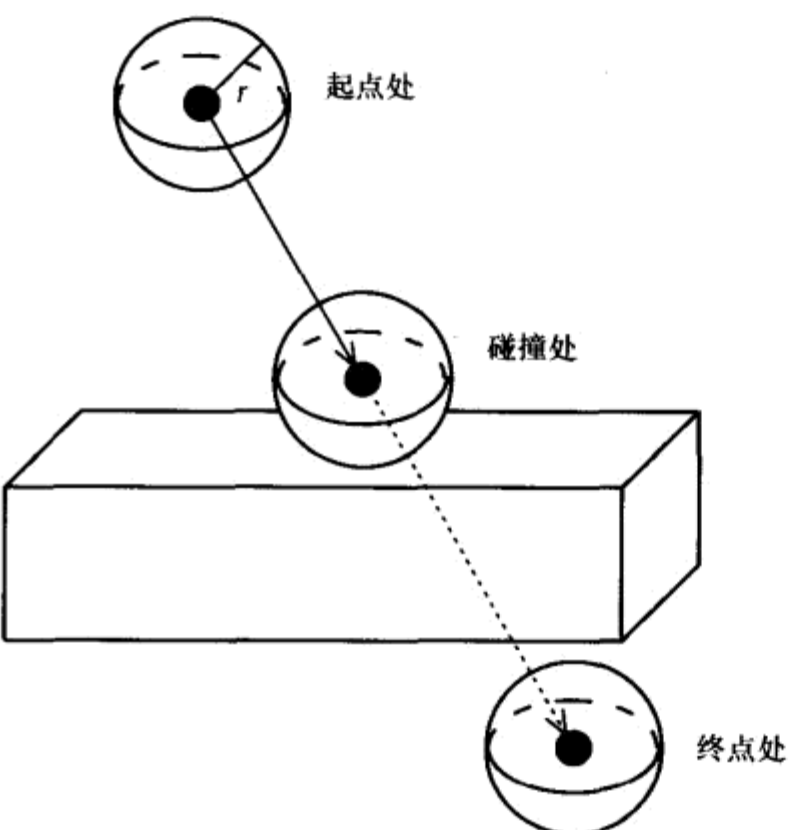


图 4.1.2 动态的“球和场景”测试

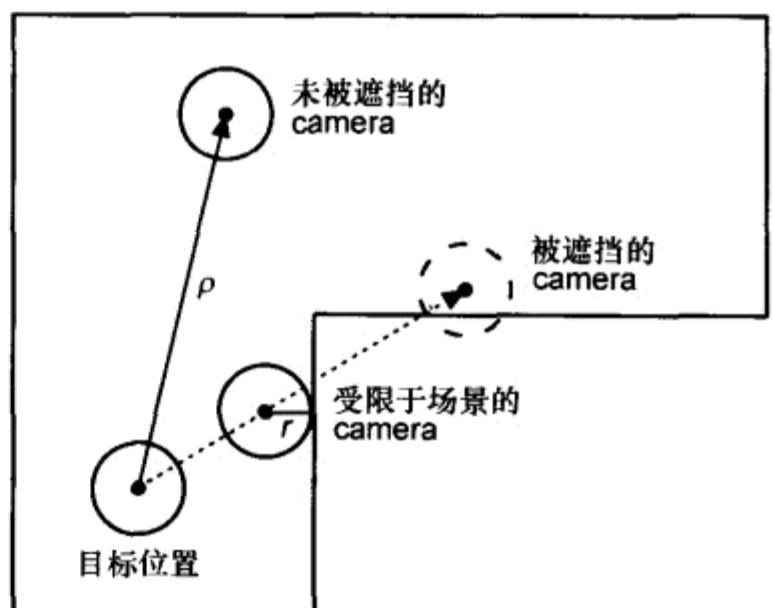


图 4.1.3 虚拟 camera 碰撞

看起来，该算法的效果就是 camera 跃前而避开了场景中所有的障碍物和遮断。当障碍物被抛诸脑后（换句话说，在目标和 camera 之间的线扫掠球体是畅通无阻的），此时 camera 的弹簧机制会慢慢返回正常距离 ρ 。此方法有效地将 camera 限制在场景中，但付出了帧与帧之间的连贯性作为代价。如果玩家绕着房间中央的圆柱行走，camera 会为了避免穿透场景几何体（即圆柱）而跃前到圆柱的另一侧。这一跃前造成了运动的不连续，在一瞬间就破坏了 camera 的平滑运动。在稍为复杂的环境中，会存在多个较小的障碍物，此类的中断将更为频繁地发生，这潜在地使玩家失去方向感。

我们可以做一项改进，即给动态球测试中的 r 取较大的值，以便在 camera 与障碍物之间发生碰撞前检测出障碍物，并且随着时间的过去插值使 camera 前进。虽然并不能消除中断，因为向前插值的 camera 仍然可能在撞到场景边界时被强制发生跳跃——而显示场景的外部哪怕只有一帧也是不允许的。但是本方法在很多场合确实使 camera 的运动得到了平滑。由于理论上能发生的最大幅度的跳跃就是幅度为 ρ 的跃前，我们也可以通过设定最小化理想球坐标的 ρ 分量而将中断减至最低，例如从 camera 的理想距离降低至零距离。

4. 物理 camera 碰撞

第二种处理 camera 的方法将 camera 看作是实心的固体，通过滑动行为解决碰撞。由于使用了通常在解决玩家物理的碰撞时使用的算法 [Melax01]，我们将此方法称为物理碰撞模型。若 camera 的边界球与场景的外部相交，则从 camera 的上一个有效位置到当前无效的位置之间作球体扫掠。若测试结果与碰撞几何体上的某多边形表面相交，则将球体沿该表面滑动 camera 剩余应移动的距离在该表面上投影下来的那段距离分量。然而这一滑动矢量可能又与场景几

何体上的其他多边形相交，因此需要多次重复这样的碰撞查询，直到 camera 移动完毕，或查询次数达到上限为止（见图 4.1.4）。

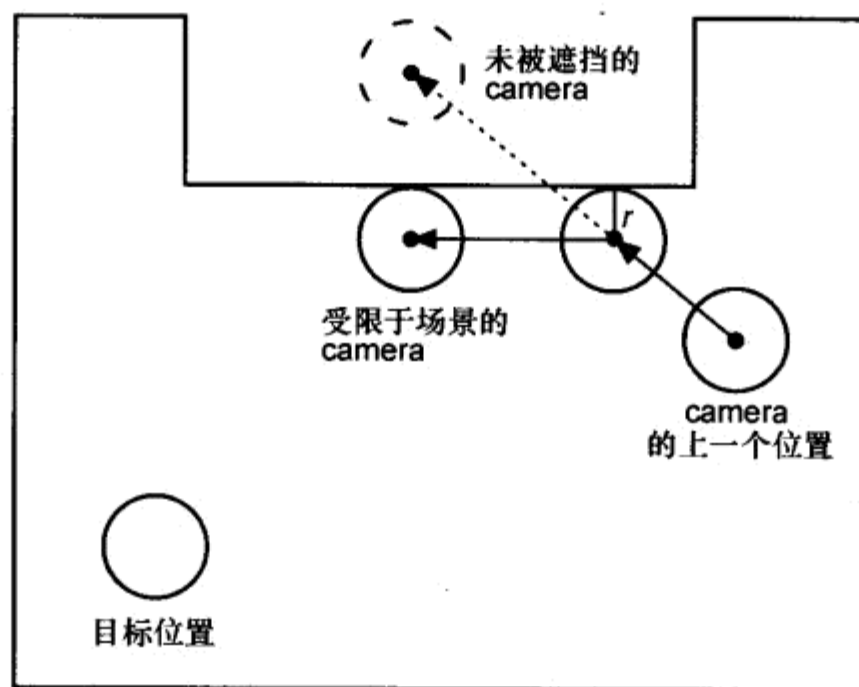


图 4.1.4 物理 camera 碰撞

看起来，这样的 camera 似乎是被场景边界“推开”了，沿小型障碍物滑动，远离大型障碍物时则慢至停止。由于 camera 总是参考上一个位置而被限制在场景的边界中，camera 在世界坐标系中的位置是连续的。若是像之前举过的那个例子一样，玩家绕着圆柱行走，camera 会采取沿圆柱的边界进行滑动的策略，避免与场景几何体发生交叉。

物理模型的一个缺陷是，滑动计算可能会改变 camera 的经度和纬度，因而潜在地干扰用户对 camera 的控制。若用户故意将 camera 朝一个大型障碍物所在的方位旋转，可能会发现 camera 停滞在墙边，因而不能自由操作。这在玩家身处狭窄空间，而又希望操纵 camera 观察特定方位的时候很可能会造成困难。为了减少问题的发生，我们可以在碰撞发生时暂时性地缩小 camera 的理想半径，当从碰撞中恢复的同时增大，使其恢复原来的值。

4.1.3 Camera 遮断

现在我们的 camera 已不会超出场景的边界，但玩家的视野仍然可能受到介入的几何体的遮掩。这种情况被称为 camera 视野的遮断。之前描述的简单的虚拟碰撞模型会自动处理这种情况——将 camera 插前到最近的无遮掩的位置，但物理模型需要单独处理遮断的情况。以下让我们为此问题提出一个解决方案，分三步：检测遮断、寻找新的无遮掩的视野、寻路。

为了检测 camera 是否在某一帧被遮断了，可以从 camera 往目标使用一次动态球测试，这会迅速判断出是否存在某个阻挡了视线的碰撞几何体。不过在实际运用的时候最好是使用更可靠的测试——将地板或天花板上细小的遮断忽略，从而判断玩家角色的综合可见性。为此，可以从 camera 往目标附近不同高度的位置，使用两次或更多次的动态球测试。若其中任一次测试产生了没有碰撞的结果，就认为所给定的视野是开阔、无遮断的。

检测出当前视野中的遮断后，我们必须为 camera 找出新的不受遮断影响的目标点的球坐

标——位置尽可能接近原先的位置为好。所使用的搜索方法的细致程度 (Level Of Detail, LOD) 依赖于环境的复杂度。若环境相对简单、正交, 又没有倾斜的地面或其他斜面, 我们就可以沿某一维方向进行搜索, 例如沿 camera 的球坐标位置的经度分量方向。若游戏中的情况更为复杂, 例如真人大小的角色沿着崎岖不平的多岩石的场景行走, 我们则需要采用二维搜索模式, 以便一并将 camera 的纬度考虑进来。

如图 4.1.5 所示, 对于一维的情况, 我们可以有效地通过一种先宽后窄的搜索找出为了在视野中消除遮挡所需对经度做的最小改变。在宽步长的阶段, 我们以较大的步长对经度进行修正, 找出最接近原先角度的无遮角度。然后在窄步长的阶段, 以较小的步长来回微调结果。这样找出了 camera 的精确的目标位置。但是由于计算过程较为复杂, 需要对结果进行暂存, 并尽量重用, 只要结果仍然有效——也就是说, 直到暂存起来的目标位置已经到达, 或由于玩家接下来的运动而变成显示无效的位置为止。

对于二维的情况, 有一些搜索模式可供选用。简单地扩展宽窄搜索法, 可以产生一张组织的测试坐标网格, 从经度/纬度的偏移量为 $(0^\circ, 0^\circ)$ 开始, 渐增地扩展到最大的偏移量 $(\pm 180^\circ, \pm 90^\circ)$ 。当宽步长的搜索找到无遮掩的视角后, 将结果来回微调。首先将纬度朝 0 微调, 然后是经度。德国马德堡大学的 Helbing 和 Strothotte [Helbing00] 建议采取螺旋的搜索模式, 从偏移 $(0^\circ, 0^\circ)$ 起用渐渐增加的步长螺旋形地打开。

如果在被遮蔽的 camera 与其目标之间存在无遮视线, 则只需要将目标点作为 camera 的新理想位置, 并允许引擎中的弹簧系统通过插值平稳地将 camera 移到无遮拦的位置。但在许多场合, 视线本身会被介入的几何体阻隔, 因此有必要进行寻路。

作为解决方案, 我们可以使用通用的 3D 寻路方法 [Smith02], 将游戏世界划分为凸的区域, 并预先 (offline) 计算区域间彼此连通性信息 (connectivity information)。针对 camera 导航的一个较为简单的方法, 便是利用由玩家在世界中的路径提供的连通性信息。我们可以将玩家目标球的最近位置保存在一片循环的缓冲区中, 将其用作“撒过面包屑的踪迹”来引导 camera 回到未遮断的目标位置。在发生遮断的任一帧里, 若经过计算的 camera 目标点不在视野中, 则先找出“撒过面包屑的踪迹”上在 camera 视野中但距离最远的那一点, 接着将中点作为 camera 的理想位置。一旦目标点本身进入了视野, camera 就可以脱离这踪迹而直接移向目标点。

见图 4.1.6, 该算法使 camera 能够跟随玩家通过门口进入另一个房间。结果产生的路径并不如完全 3D 寻路算法所能给出的路径那样平滑, 因为后者能够预先计划 camera 的路径并利用样条曲线使结果线段平滑化。但是, 该算法在如玩家快速通过复杂环境那样的动态情况下效果依然良好。有时候, 该方法可能让 camera 位置过于接近目标玩家本身, 以致 camera 的朝向频繁出现翻转。在这样的情况下, 有必要使 camera 远离目标玩家以满足距离

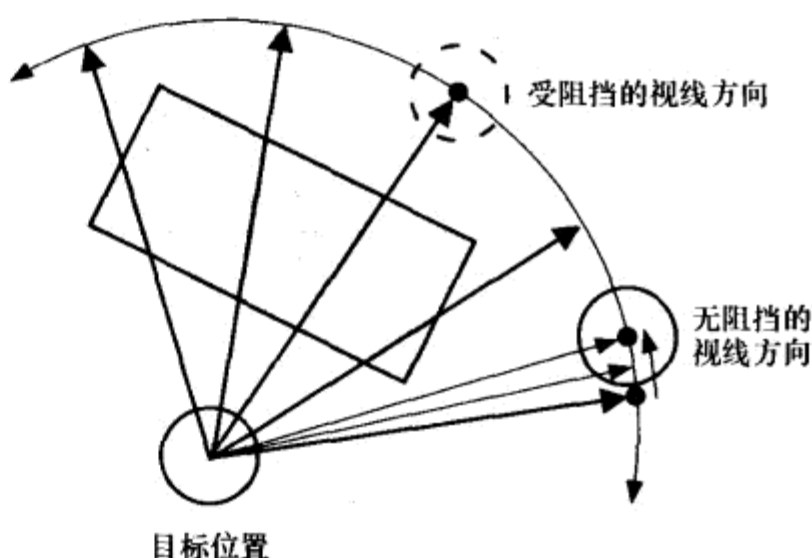


图 4.1.5 在一维方向上寻找无遮掩的视野

的下限。

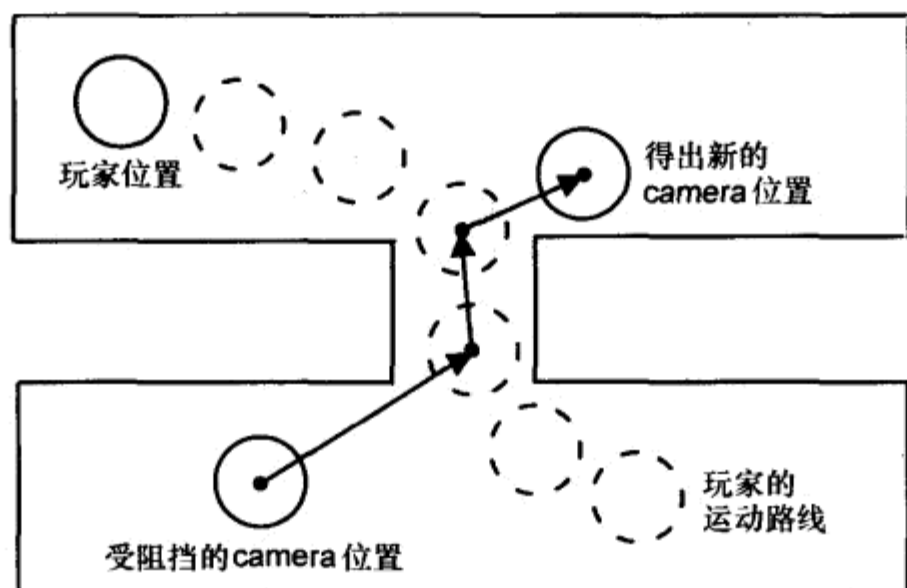


图 4.1.6 面包屑寻路

4.1.4 简化场景

这点一定要记在心里，有时候通过简化 camera 在场景中的表示，或通过简化待渲染的场景本身，比起单纯依赖漫游的逻辑，能更有效地计算 camera 的状态。在一个充满障碍物的高密度场景中，camera 若是频繁调整位置会使用户感到混乱，因此若能避免此类状况，gameplay 会更加平滑。

简化 camera 环境的途径之一是在关卡设计 (level design) 的阶段就对潜在的遮断几何体做出是否能将其从 camera 的必经之路上移开的判断。另一途径是允许 camera 透视某一些实心物体，也就是当这些物体阻挡玩家的视线时将它们半透明地渲染。因为减少了计算 camera 的无遮拦视线时需要考虑的物体数量，半透明地渲染对象有效地降低了场景的复杂度。但半透明应被合理且有限地使用，以免破坏游戏世界的真实感。

4.1.5 结论

本文介绍了一些方法作为实现动态的、用户可控的第三人称 camera 系统的基础。也描述了一些有关 camera 和场景边界体设定以及遮断的棘手问题的解决方案。任何一款游戏都应当在 camera 运动的平滑、用户控制的弹性、复杂障碍场景的漫游这三者间找到自己的平衡。总之，我们希望本文提出的想法能在不同的游戏设计中得到运用。下一小节列出的参考文献旨在提供本文中所涉及的数学的详尽背景知识，以及研究和寻找灵感用的参考读物。

4.1.6 参考文献

[Helbing00] Helbing, Ralf, and Thomas Strothotte, "Quick Camera Path Planning for Interactive 3D Environments," available online at

<http://isgwww.cs.uni-magdeburg.de/~helbing/publ/quickplanning-sg00.pdf>.

[Melax01] Melax, Stan, "BSP Collision Detection as Used in MDK2 and NeverWinter Nights," available online at www.gamasutra.com/features/20010324/melax_01.htm, 2001.

[Moller02] Akenine-Möller, Tomas, and Eric Haines, *Real Time Rendering, Second Edition*, A K Peters, Ltd., 2002.

[Smith02] Smith, Patrick, "Polygon Soup for the Programmer's Soul: 3D Pathfinding," *GDC 2002 Conference Proceedings*, 2002.

[Treglia00] Treglia, Dante, "Camera Control Techniques," *Game Programming Gems*, Mark DeLoura, Editor, Charles River Media, Inc., 2000.



4.2 叙述战斗：利用 AI 增强动作游戏中的张力

作者：Borut Pfeifer, Radical Entertainment

E-mail: borut_p@yahoo.com

译者：肖罡

审校：李劲松

真正的游戏迷一定玩过 AI 部分是由大量脚本控制的游戏。这类游戏，设计者完全控制了游戏的节奏，玩家必须按照固定的模式一步步走下去，所以通常会有若干个精彩的游戏高潮部分。玩家可能会觉得某个这类游戏的节奏很新奇——但仅仅是第一次玩的时候；随着玩的次数的增多，玩家会清楚地知道什么地方会出现什么敌人以及他们要做什么，这种可预见性也就极大地降低了玩家的乐趣。此类由脚本完全控制的游戏通常不给玩家控制的自由，同时设计者想要设计出具有出色组织节奏的惟一方法就是事前做好各种可能发生事情的脚本。

如果说游戏完全控制了节奏会让玩家厌倦，那么走到另一个极端会怎样：把对游戏节奏的控制几乎完全交给玩家。如果这样，玩家很有可能连游戏当前的目标都不知道，毫无目的在游戏中游荡，会得到很无趣的体验；在游戏的可玩持久性这个角度上来看，该游戏反而丧失了高度有效的节奏。

理想地，最好能有某些机制让游戏设计者根据玩家的进度来调节游戏的节奏。这些调节机制不仅仅是由游戏设计者决定的，而且还要考虑到玩家当前的经验和技术水平。这样一个更加动态的环境通过提高或降低戏剧张力使游戏的可玩性始终处于一个会让玩家觉得享受的状态。

这篇文章讨论了如何创建一个 AI 系统来调节动作类游戏的难度和节奏。这个系统除了把高度节奏化的游戏元素、可玩性和更自由的游戏控制结合起来，还可以使游戏设计者更容易创作出有好的节奏的游戏——这是因为这个系统只需设计者对特定的一个地形给出其大致的难度，而不像大多数其他游戏那样，必须把敌人的出现位置、出生点、武器、时机问题等很多方面都事无巨细地实现编好才能达到同样的效果。适应玩家的技术水平同时也避免了另外一个普遍问题：设计者按照他们自己，而不是典型玩家的技术水平来调节游戏的玩法，这样会使中等水平的玩家感到很困难。文中描述的系统可以适应玩家的技术水平，也就实现了当游戏节奏改变时难度的平滑过渡。

4.2.1 戏剧张力

如果对高中（美国）的语文课还有印象，你也许会记得老师曾画过类

似图 4.2.1 的图。这张图表示出的那种先高后低的戏剧张力可以在很多地方找到，比如消闲小说、电影、电子游戏甚至一场篮球比赛。戏剧张力可以被想象成让观众参与的程度，进一步说，是复杂度的一种表示：观众已知多少信息，未知多少。据个例子来说，随着一篇玄异小说剧情的展开，越来越多的关于谁是凶手的问题被提了出来，新出现的信息同时带出更多的未知信息。在一个游戏中，这个所谓的未知就是玩家是否能玩成功。张力会随着剧情的进展逐步提高，直到达到一个高潮顶点。顶点出现的位置有很多可能，当通常会在靠近结尾的时候出现，随之张力会逐步减少。在高潮过后的短暂阶段，余下所有的悬念都会被解开，这也就是所谓的结局。

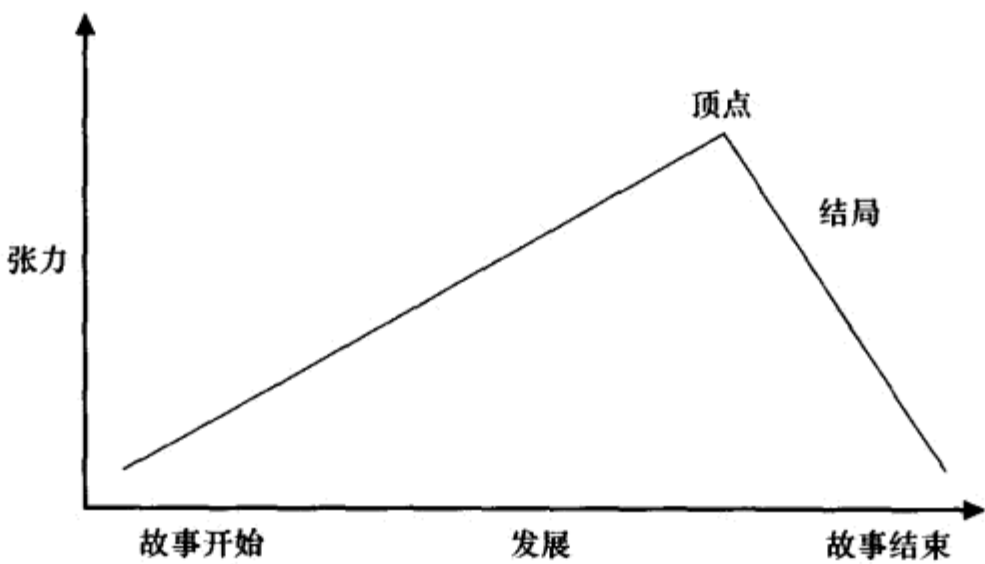


图 4.2.1 基本的戏剧结构

实际上，一个故事的戏剧结构，无论这个故事是一本书、一部电影还是玩家在一个游戏中的体验，都是千变万化的，但那种总的张力不断增加，一直到达定点的趋势线依旧有效，只是在其中会有很多个局部的高低起伏。在一个游戏中，局部的高点可以是有难度的挑战，比如关底之战；而局部的底点可以是低紧张程度的阶段，比如一个新关的开始（如图 4.2.2）。

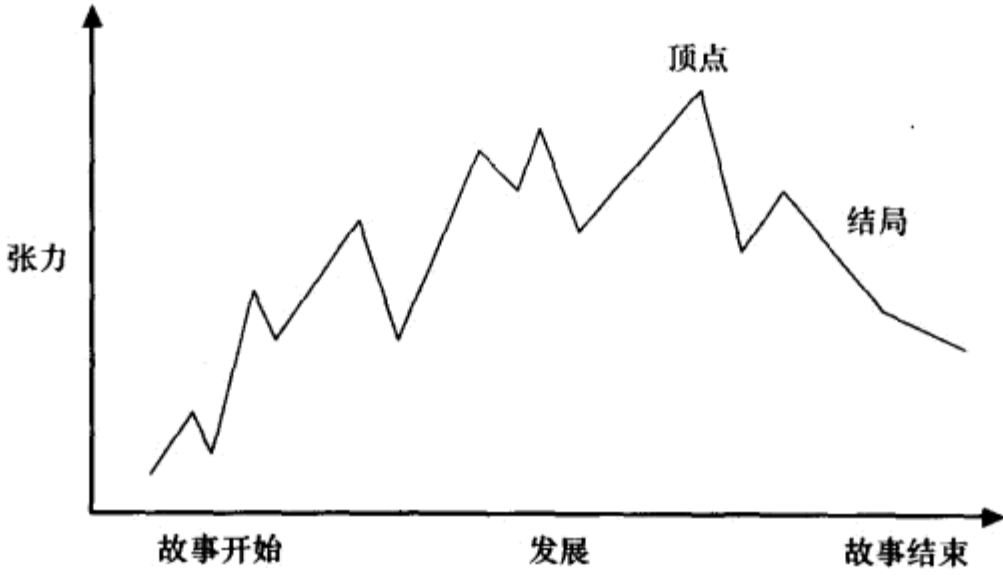


图 4.2.2 更真实的戏剧结构

Brenda Laurel 在她的“*Computers as Theatre*”一书中，提出了用戏剧模式作为用户接口的设计规范[Laurel 91]。这主要是因为戏剧模式侧重于情节，同时也定义了交互性。因为事件的选择、安排都是用来表述要被强化的情感，所以情节加速压缩了时间。组成戏剧的各个独

立事件是紧密地围绕在中心情节上的。

这种高低起伏的戏剧表达方式与传统的平铺直述式的主要区别是：平铺直述式主要依赖于描述，个体事件的描述通常占据很多时间，被描述得非常具体，而事件之间可能本质上并没有什么内在联系，只是罗列在一起。

戏剧张力的概念在游戏中并不是指故事情节的发展，而是指玩家的技术水平和游戏当前设定难度之间的关系。在其心里最优体验（或舒适度）领域中，Mihaly Csikszentmihalyi 研究了人在其技术水平、所要完成任务难度相互作用下产生的不同舒适程度体验。好的舒适度会让人忘记时间等其他事物，全神贯注于当前工作中，游戏想要玩家觉得好玩这个目的和那种舒适度的概念是相同的。

很明显，一个玩家在刚开始玩一个游戏时水平是很低的，随着玩的次数增多，玩家的技术就会提高，如果此时游戏的难度还保持不变的话，玩家就会感到很没意思。游戏中的难度通常会逐渐提高，但这种提高的速度和玩家技术水平提高的速度很难保持一致。因此，玩家玩游戏的感觉不外乎是图 4.2.3 中的 4 个点：或者感到厌烦，如果游戏难度没有跟上他水平提高的速度；或者相反，感到很沮丧，如果自己学得不够快。

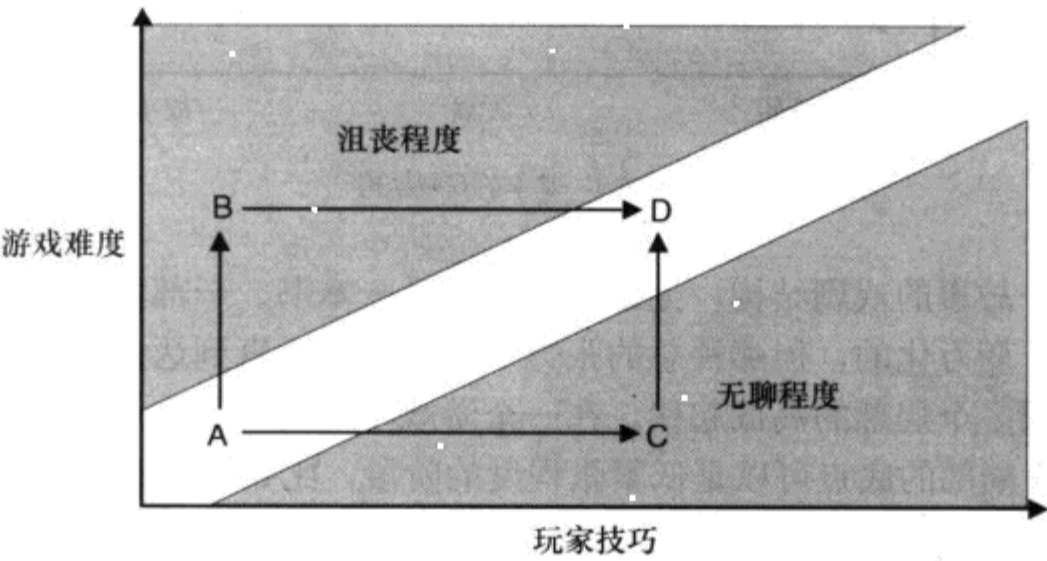


图 4.2.3 玩家游戏体验转换图

在点 A，玩家刚开始玩这个游戏，对游戏知之甚少，所以此时游戏的难度应当是最低的。如果一开始就给很大的难度，玩家会因为水平不够而很沮丧（也就是图中的点 B），假设玩家没有因此而退出，那么最终随着水平的提高也会走到点 D。

相反，如果游戏的难度增加得不够快，玩家就会感到厌烦，认为这个游戏太简单（点 C）。随着游戏难度的增加，玩家最终会因为能发挥自己的水平而得到乐趣（点 D）。

游戏节奏（不管是不是用脚本控制）的目的，是让玩家始终处于图中白色的区域（舒适区），也就是游戏难度和玩家技术水平在一定范围内匹配的区域。但是，在不同玩家学习的能力、不同难度以及不同任务三者的交互作用下，每个玩家都会经历由厌烦（太简单）或者沮丧（太难）回到舒服（匹配）的过程。

在图 4.2.3 中，没有戏剧高潮的概念。只要玩家的感觉在舒适区（白色区域），他的技术水平一定与游戏当前难度大致相当。因此，当玩家应用所学技巧去解开难题从而获得满足的体验，和难度与技术水平相匹配的关系是一样的。

有清晰的目标是让玩家获得舒适体验的关键之一，所以有效地达到戏剧顶点的方法是和行动目标相依相辅的。在一个关底，降低张力（难度）并强调一下玩家当前所学，会让玩家有很强的成就感。但是，如果难度降低太多，就会让玩家觉得很无趣，从而产生不舒适的体验。

在人工智能领域中一个正在成长的方向就是用人工智能来控制交互式叙述的戏剧结构。比如，Mates 和 Stern 在 2003 年的游戏开发者会议上描述的一个智能系统 *Facade: A One Act Interactive Drama* [Mateas03]。虽然当前很多这类系统加入了一些很高的次要目标，比如自然语言处理，但不管是怎样的交互式环境，其主要目标是共同的：用控制环境中发生的事件来微调用户的体验。这类系统就像个调度者，把各种事件以合适的节奏安排出来去取悦用户，在依旧遵循戏剧结构的前提下，按照用户的当前技术水平来合理调整难度，使用户始终处于愉悦的状态。

4.2.2 系统概述

建立上述所说的系统通常有以下几个步骤。

(1) 分析在游戏中哪些元素是可以随着玩家水平的进步动态地改变的。这些元素就要被用来调节游戏的难度。

(2) 对每一个可动态调节的元素，设计出一种度量方法来描述其难度。

(3) 找出能控制总体难度的方法，也就是说，任给游戏的一个片断，能有方法判别出它比其他游戏片断是难是易。

(4) 定义出一个尺度能衡量出在玩游戏的过程中任何一点玩家的技术水平。

(5) 建立游戏可控的动态元素和玩家最终体验难度之间的关系。这种关系决定了如何用这些元素来调节游戏的难度，也就是如何根据玩家的水平来合理组合它们。

4.2.3 设计者的控制部分

设计者有两大类输入给这类 AI 系统：一类是控制游戏中每种地形相对于其他地形的难度；一类是给定游戏中每一个动态元素的难度评价函数。

对于设计者的每一种输入又都有两种方式来实现：第一种，也是最直接的一种，就是在游戏生产过程中，设计者就给系统暗示了这些大概应该是什么；第二种方法是创建一种在线机制，在玩游戏的过程中能动态评估出如何最好地定义这些控制参数。

因为大多数游戏都有一些空间探索，一种很简单的方法就是把游戏难度和地形的类别结合起来。当玩家进入到某一类地形时，游戏利用该地形的特有属性来设置难度。每一种地形有难度级别（比如说 5 级，从最简单到最难就够了），再结合玩家的技术水平，就可以定出目标难度。每一种地形还有一些可供动态元素用来进行难度调节的附加数据。例如，一个地形（用 trigger volumn 来表示）可能会有可出现敌人的种类（每一类有自己的难度级别）以及出现的地方这些数据。有了他们，游戏就可以组合不同种类的敌人、数量以及出现地点，来控制难度。注意，如果游戏不允许在玩家的可视范围内动态创建敌人，设计者可以把敌人的出现地点放在玩家的视野外，比如墙拐角或者门后。

另一种方法是按顺序把游戏的任何一点给定一个相对的难度，比如一种设置方法就是根据玩家在当前局所处的进度设定难度。当快靠近关底时，游戏自动突升难度，然后结束前再降低。如果玩家在任何一个地方花了很多时间还是没有过去，难度就要降低（提供一个自然的戏剧张力低谷），帮玩家过了这个难关。

第二类游戏设计者要给系统的输入是各个动态元素的难度评估。比如动态元素是出现不同数量和种类的敌人，那每个敌人就可以有固定的难度级别（为常量）。定义一个计算方法通常是不容易的：一个设计者的启发式评估会补偿很多潜在的因素（比如武器杀伤力、射程、精确度、移动和攻击速度、AI 策略等等）而不用增加整个系统的复杂度。如果各个动态元素所属数据变化得很频繁，也许就需要一个机制来定义元素的难度，但复杂性往往会让难度很难调节。一个游戏中可能发生的场景是：敌人突然得到什么东西（比如武器、加速、盾牌等等）使其能力大增（相对玩家而言），此时游戏则需要根据设计者给定的子元素来计算给定动态元素的难度。

4.2.4 难度计算

很多游戏会根据玩家的技术水平来调节难度。比如在赛车游戏中，紧跟着领先车的赛车常常会得到一个加速。要应用调节难度的方法类控制节奏，我们必须能够实时评估当前玩家的技术水平。如下元素是在动作或战斗游戏中，我们可以用来构造难度的调整模型。

- **打败当前敌人的时间：**从玩家刚接触到这个敌人开始计时，到敌人被消灭时截止。
- **敌人难度级别：**设计者可以把敌人按难度分级，比如从 1 级到 10 级（最高难度）。
- **同时出现的敌人：**一场战斗中同时出现的敌人的数目。

当然还有很多别的因素（比如玩家的准确度）也可以用来控制难度，但最重要的是能找到能反映出玩家技术水平的核心关键元素来进行一个简单评估。玩家如果使用一个威力强大的武器，即使精确度很低，仍然可以迅速打败敌人。我们先只考虑这些基本的元素、主要的时间，一旦敌人被打败了，我们就可以自动考虑更多的因素（比如玩家保留利害武器的能力、他的准确度等等）。

最终，我们希望能有一个简单的公式，当给定特定玩家特定时间，公式能返回一个数值来代表难度。数值高就表示当前的难度高，反之就是低。直觉上讲，高难度说明玩家水平高，他面对的敌人难度级别也高；如果玩家要花很长时间（超过平均时间）来打败敌人就说明玩家水平不行。

$$\text{玩家技术水平} = \text{敌人的难度} / \text{打败敌人所需的时间}$$

理想地，在玩一个游戏的过程中，通过上边这个公式我们会得到一组递增的数。我们可以用原来度量敌人的难度的方法（从 1 到 10），来度量玩家的技术水平（1 表示最低，10 表示最高）。难度调整系统可以利用这个数来反向选择敌人的难度（所以我们最好让这两组数在同一个值域内）。要做到这一点，我们就要把上面算式中的时间乘以一个常量，这个常量表示这样一个基本的难度单位：一个典型新手（技术水平等于 1，第 1 次玩这个游戏）用最简单的武器装备打败一个难度最低的敌人所花费的时间。

最后一个要考虑的是当前玩家要同时面对的敌人的数量。很显然，同时打败 4 个敌人要

比分别打败他们难得多,所以,敌人数量也是一个可以影响上面公式的因子。要注意的是,在不同的游戏中,同时出现敌人的数目的增加对难度的增加影响程度是不一样的,所以设计具体游戏时要根据游戏的性质分配该因子在公式中的权重。

延续上边的例子,如果单独打败一个敌人玩家的技数水平需要是1的话,两个敌人就需要1.125,3个敌人就要1.25,4个就要1.5。如果玩家同时面对3个敌人,打败其中任何一个都会比单独打败他要难(难1.5倍)。

如何得到这些数字就和游戏的设计机制有关:这个游戏把同时对付两个敌人给设成什么样的难度?很多游戏给玩家提供特定的武器或者工具可以同时消灭很多敌人,但有些游戏规定玩家一次只能打一个敌人。当这些东西都可以由主观决定时,要想设定那些数值(1.125、1.25、1.25),最好找一些真人通过玩游戏的方式来估算:一个敌人多大难度,两个敌人多大相对难度,依此类推。当使用“同时出现敌人数目”这个因子时,也可以把在这个组中的其他敌人的影响也考虑进去,但实践证明,这样所增加的复杂度和所得到的稍微精确一些的玩家技术评估不成比例。

最后的公式应如下所示。

玩家的技术水平 = (敌人的难度 × 同时出现敌人的数目系数) / (打败敌人所花时间 × 单位难度)

这个公式可以很好地给出在游戏中任何一点的难度估计,游戏必须跟踪记录玩家近期的游戏经历。这样一个集成的难度评估系统其实就是把各个评估方法加权平均。

这样,我们可以通过连续评估用户的技术水平而得到一组数字,平均这组数字就可以得到玩家在最近几分钟的平均水平。更好方式是用时间的远近来加权平均那组数字,因为玩家可以为了要一个暂时的低难度而故意降低自己的水平来欺骗计算机。为了防止这种情形发生,我们可以只简单地统计最近5分钟的数字,在这么短的时间内玩家行为不大可能变化很大。

4.2.5 难度调节

有了上节得到的玩家技术水平估计为基础,游戏就可以决定目标难度。具体的方法是,根据当前游戏进度应该有的难度,再结合玩家的技术水平,就可以得到目标难度。具体可以用很多种方法来实现:如果游戏通过定义地形的方法来反映给定难度,那么就可以先周期性地估算出玩家在特定地形的技术水平,再和目标难度进行比较以做出难度调整;但如果游戏使用一套特殊机制来定义整体难度如何变化(比如考虑玩家已玩完游戏的比例或者已经完成了多少目标),那难度就要由这个机制产生了,即使这样,选择和创建必需动态元素的方法也是一样的。

用地形来定义难度,设计者要从1(最简单)到5(最难)分配一个数值给一个地形。玩家的技术水平可以作为初始的目标难度(通过映射两者的取值范围)。每一个地形的一个难度值,都对应着一个调节因子,地形对难度的调节就是用这个调节因子去乘以目标难度。比如说,如果地形难度值为1,目标难度就减半;如果是5,就加倍。

至于调节因子的形式,可以直接让设计者指定调节因子,也可以用间接的方法。间接的方法有两个好处:一是有限的离散难度只会让设计者更直接地体会到最终的难度大概是什么样子的(比如5个离散值:非常容易、容易、正常、难和很难);二是间接的方法可以让设计

者把调节因子由数值形式转到描述性形式。例如，如果设计者觉得调节因子等于 2.0 没有使游戏变得足够难，可以把调节因子改为描述性的“非常难”，那样就不用在游戏里四处为具体应该是什么值而头疼了。

当玩家已置身于一个地形中，那个地形必须根据目标难度选择合适的动态元素。具体的方法和上边所说的调节因子产生的方法类似。例如，如果玩家已经完成了一半的目标任务，就应该用 1.5 去乘以玩家的技术水平从而得到目标难度。

现在游戏已经有了一些可供选择的动态元素（每一个都有自己的难度级别），以及一个目标难度。游戏要考虑如何安排这些元素来给玩家更好的体验，同时要组合这些动态元素以满足给定的目标难度。比如，可以用一个很难的敌人搭配一个弱的敌人，或者两个同样较难的，或者 4 个弱的，选择的方法要考虑美学，同时要考虑强调游戏的主题，随机的选择方法可能不会产生好的游戏体验。

4.2.6 系统评价

应用这样一个系统之前，有几个设计上的问题必须要考虑清楚。难度和玩家体验之间的关系是用任意值表示的输入参数（比如 1 到 10 可以用来表示相对难度），要调试整合这些参数到合适的状态是很难且很花时间的，所以，通常最好整个机制和那些关系越简单越好。理想的是，通过设计者的几个基本输入就可以给玩家创造出好的动态体验。

对于这些用来微调交互体验基于规则的系统，一定要注意不要把内部的机制暴露给玩家。交互式的音乐就是一个很好的例子。如果游戏根据当前游戏场景立刻调整了背景音乐，玩家就会很快发现这个规律，从而会利用它。比如，敌人一接近，某种音乐就响起，玩家就会利用这个音乐来预测（像发现埋伏什么的），这样玩家不会有好的游戏体验了。类似的，难度调整系统一定要考虑到玩家之前的游戏表现，这样就不会被玩家轻易骗掉（比如，玩家故意花很长时间来杀一个敌人而使游戏难度级别降低），要做到这一点，就要参考玩家之前很长一段时间的表现来调整难度。

这个系统的目的是克服以往这类游戏的很多弊病（比如重玩性差），从而创造出有非常好的节奏感的游戏。通过控制很少几个输入来控制难度将使这件工作变得容易，同时会产生流畅的游戏体验，即使在动态复杂的环境也一样。这也同时避免了设计者要事先实现几个不同节奏、靠玩家手动选择游戏版本（比如，容易、正常和难），从而帮助玩家在玩游戏的过程中创造出属于他自己的难度。Bioware 的 *NeverWinter Nights* 中的编辑工具允许玩家设定敌人（出现的方式和次序），虽然战斗也都是由各个元素决定的，但并不考虑玩家的因素，所以整个系统并没有考虑到玩家以及以往战斗的难度。本文所描述的系统就考虑了这些因素。

当这个系统被用在战斗类游戏中时，用动态的模型调整玩家的体验这个概念是很有用的。与本文中的系统目的类似，一些常规软件也试图通过预测用户的目的来改进用户使用体验，比如一些文字处理软件中，当用户敲入头几个字母，软件会根据自动添完其他部分，这样的系统可能开始会和用户的习惯有冲突（猜错了用户的真正目的），但用久了，就会和用户的习惯融合起来。回到游戏上，也就意味着玩家会获得更享受的游戏体验，而不会时常觉得节奏太快或太慢。

可能因为开发队伍为了赶进度或者预算上的原因，这些原则还没有被广泛应用。但是，

随着游戏业发展到更广阔的市场,游戏设计者会为了跟上玩家的想象力不得不让玩家获得更加协调的戏剧体验。

4.2.7 结论

这篇文章描述了可以应用在动作类游戏 AI 系统,它能够根据玩家的近期表现来调整难度。根据玩家以往打败敌人的时间来估计玩家的技术水平,进而调节要出现敌人的难易程度,这样设计者就可以给所有水平的玩家提供适合他们的难度。

4.2.8 参考文献

[Mateas03] Mateas, Michael, and Andrew Stern, "Façade: A One Act Interactive Drama," *Game Developer's Conference 2003*. (译者注: InteractiveStory.net, www.quvu.net/interactivestory.net/papers/MateasSternGDC03.pdf)

[Csikszentmihalyi90] Csikszentmihalyi, Mihaly, *Flow: The Psychology of Optimal Experience*, Harper Collins, 1990.

[Laurel91] Laurel, Brenda, *Computers as Theater*, Addison-Wesley, 1991.



4.3 非玩家角色决策：处理随机问题

作者：Karén Pivazyan, Stanford University

E-mail: pivazyan@stanford.edu

译者：肖昱

审校：李劲松

游戏 AI 的很大精力都花在计算非玩家角色如何在决策点决定自己的行为上。这些行为包括从寻路、选择建造合适的单元直到选择攻击方式等。这样的决策对游戏以后的发展都是有影响的，例如，在某个时间，如果我们选择建造了步兵的话，就可能让下一步发动一场空袭的计划泡汤。

因此，仅仅让计算机考虑出当前最好的决策是不够的，我们必须要考虑每一个行动对以后的全局的影响。实际上，我们的目的是要能算出从当前状态开始到最终目标为止整个过程的最佳决策序列。在确定的环境下（比如只有静止障碍物的寻路过程），我们可以用像 A* 这类搜索算法来做到这一点。

但是，假设有一个疯狂的魔法师在念咒语，产生的结果不是惟一的，而是几种可能之一。此时这种随机性就破坏了 A* 算法的要求，如果要强行应用的话，最好的情形是要做很多修改工作，最差的情形是 A* 会给出错误的答案。问题是，在游戏中随机现象随处可见：攻击的次序、咒语的结果（时间、地域的影响）、技巧的应用（如躲藏、偷窃），这些都可能产生随机的结果。我们怎么处理这样的情形呢？

在这篇文章里，我们描述了一个算法能解决随机条件下的多步决策问题，最基本的算法叫做动态规划（Dynamic Programming，缩写为 DP）。已经有很多个基于 DP 的算法被成功地应用到俄罗斯方块（Tetris）和西洋双陆棋（Backgammon）等问题上。

4.3.1 概要

动态规划算法能够在一张随机地图上找出两点的最短路径。随机地图是相对于确定地图（固定顺序的一系列操作一定会得到相同的结果）而言的，因为随机地图上每一步操作的结果都是随机的，所以即使是同样的操作序列也会得到不同的结果。

为简单起见，这篇文章仅限于 2 维随机地图上的寻路问题。当然，像 A* 一样，DP 算法并不局限于寻路，我们会在最后一节讨论它的其他应用，在这里，我们只用一个角色扮演类游戏中的寻路问题作为例子。

在图 4.3.1 中, 一个喝醉的年轻人必须找到一个出酒馆的路, 酒馆中的吧台是障碍物, 年轻人从 D4 出发, 出口在 A6。因为喝得太多, 他不能很好地控制自己的行动: 如果他本意是向前走的话, 结果会有 0.5 的可能性向右拐, 0.25 的可能性向左拐。具体说来, 如果他想从 D4 向北走, 年轻人走到 C4、D3 和 D5 的概率分别为 0.5、0.25 和 0.25。任何时候, 如果撞到墙, 他要回到原来的格子。

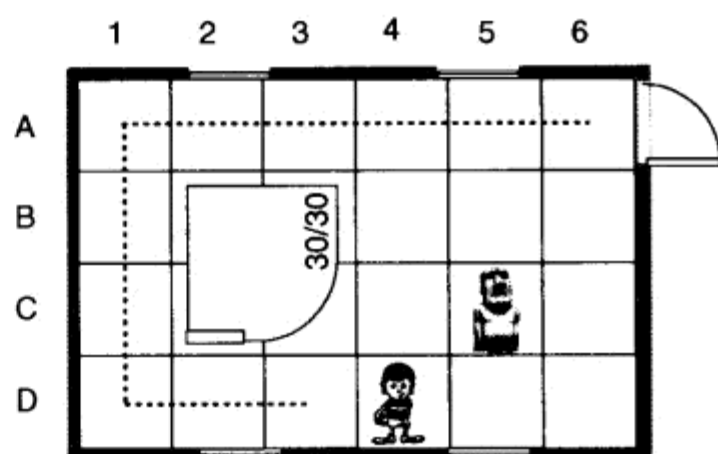


图 4.3.1 一个带有陷阱的迷宫

我们的目的是找到从 D4 到 A6 的最短路径。我们先试用 A* 来解决这个问题, 因为 A* 算法的限制, 我们必须假设所有条件都是确定的, 每一个行动都有固定的结果 (比如说, 从 D4 向西走一定会走到 D3)。问题是, 在某一时刻我们用 A* 算出了一个所谓的最短路径, 但在执行时, 因为随机因素的存在, 年轻人不会按照这个路径走下去, 当他偏离了用 A* 已算好的 “最佳路径”, 原来计算的路径也就没用了, 我们不得不重新计算, 因为 A* 不能利用上次计算中本来还有用的东西, 内存和 CPU 资源都会被浪费了。实际上, 在酒鬼的例子中, 年轻人很有可能要走遍图中大部分的格子, 因此用 A* 的话会做大量冗余计算。

除了浪费资源, 在随机地图上应用 A* 算法还有个更大问题: 会产生完全错误的路径。想象一下, 如果有个讨厌的巨人坐在格子 C5 上喝酒, 他会吃掉任何撞到他身上的年轻人。A* 算法就会天真地假设年轻人只要不走巨人在的格子, 在巨人身边溜走也行 (也就是图中的 C4、B4、B5 或者 D5、D6、B6)。实际上, 那个年轻人如果选择刚才所说的任何一条路的话, 他有超过 50% 的概率会被吃掉, 最好的路其实是左边那条 (图中虚线表示的)。

DP 算法就是用来解决随机地图问题的, 它很容易就会找到图 4.3.1 中问题的最优路线。算法自身非常简单, 相对于 A* 算法, 它需要更多一些数据来设置, 特别的, 它需要指明地图中的不确定概率。它也没有 A* 算法那么快, 因为在确定地图中, 从起点到终点找到一条最短路径就可以了, 而对于随机地图, 我们需要计算出每一种起始状态到终点的最佳路径。当然, 我们会引入很多优化措施来加速 DP 算法, 这些将在文章的最后讨论。

4.3.2 动态规划算法

现在我们来研究一下 DP 算法, 其间会用 A* 与它作比较, 所以我们假设读者对 A* 算法很熟悉了。《AI Game Programming Wisdom》书中有好几篇好的文章来介绍 A* ([Matthews02] 和 [Higgins02]), 如果读者想看关于 DP 算法在理论上和实际应用的严谨数学表述, 请参阅 [Bertsekas01]。

首先, 我们来了解一下寻路代价的概念, 任何一种搜寻最短路径的算法都必须有比较路径好坏的方法, 比较路径长短是一种很自然的方法, 路径的长度可以想象成每一步长度的累加和。在图 4.3.1 中, 从 D4 到 A6 的欧几里得长度为 $\sqrt{2} + 1 + \sqrt{2} = 3.8$ 。但如果地图上包含不同的地形怎么办? 穿越高山当然比跨过平原难, 所以我们应该把穿越高山的代价设为 10, 把跨过平原难的代价设为 1。所以现在我们在不是比较路径的欧几里得长度, 而是比较穿过它们的代价。实际上, 我们不应该再说最短路径的概念, 取而代之应换成最佳 (最小代价) 路径。

这不仅仅是简单的文字游戏，用路径的代价来选路允许我们可以比较和长度完全无关的路径，特别是，在图 4.3.1 中，我们可以用代价 100 来表示巨人所在格子，我们要寻找一条绕开它的路线。

一个 DP 算法需要以下数据：

- 目标格子；
- 每个格子的代价；
- 每个操作产生结果的概率表。

图 4.3.2 就是上边找路例子的有目标格子
和各个格子代价的对应图，目标格子用代价 0
来表示，一旦到达，非玩家角色就会陷在那儿
而不能有任何动作。因为算法是在寻找代价最
小的路径，我们一定要小心地设置各个格子的
代价值。如果凑巧形成了一个寻路代价回路，
算法就会陷入死循环。

还要明确没有起始格子了，因为算法要把
所有格子到目标格子的最佳路径都计算出来。
这是必要的，因为不确定性，小人有可能走到
图中的任何一个格子上，那时就必须知道从那个格子该怎么走（我们可以对这一步进行优化，
以后会讨论）。

最后，表 4.3.1 是对应上边游戏的概率表。对于这个简单的图，每个动作产生结果的概
率值都是一样的，但通常在更复杂的例子里，概率值要随着位置的不同而变化。

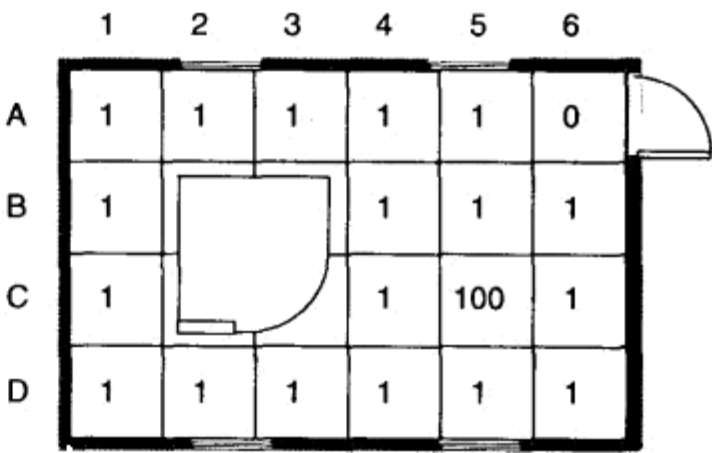


图 4.3.2 格子的代价值

表 4.3.1 动作及其效果的概率

动 作	结 果	概 率
向北	向北	0.5
	向西	0.25
	向东	0.25
向东	向东	0.5
	向北	0.25
	向南	0.25
向南	向南	0.5
	向西	0.25
	向东	0.25
向西	向西	0.5
	向北	0.25
	向南	0.25

算法

算法始终维护着一个装有代价值的数组，数组的长度等于地图的格子总数。DP 算法一
遍遍地更新数组，每个格子到目标格子的代价累计估计值被记录下来。当算法结束时，数组
中的每个元素就是每个格子到目标格子的最少代价值。我们先看看 DP 算法的伪码。

- (1) 初始化数组中的各个元素。
- (2) 对每个格子：
 - a. 计算所有在这个格子中可选择动作的代价值；
 - b. 选择一个代价最小的动作；
 - c. 把该格子的代价值设为被选择动作的代价加上格子本身的代价：

$$V = U_s + cost$$

- (3) 重复（开始下一个周期，回到第二步）。

图 4.3.3 显示了头 3 个周期和收敛到最优解时的各个格子的代价值。有几步需要说明：上述伪码第二步 a 中，每个动作的代价等于各个邻接格子的代价值与各自对应概率乘积的累加。以周期一为例，从 D5 点出发，如果执行向北走这个动作，这个动作的代价为：北边格子的值（100）乘以向北走的概率（0.5），加上西边格子的值（1）乘以向西走的概率（0.25），加上东边格子的值（1）乘以向东走的概率（0.25），最终向北走的代价为 $100 \times 0.5 + 1 \times 0.25 + 1 \times 0.25 = 50.5$ 。

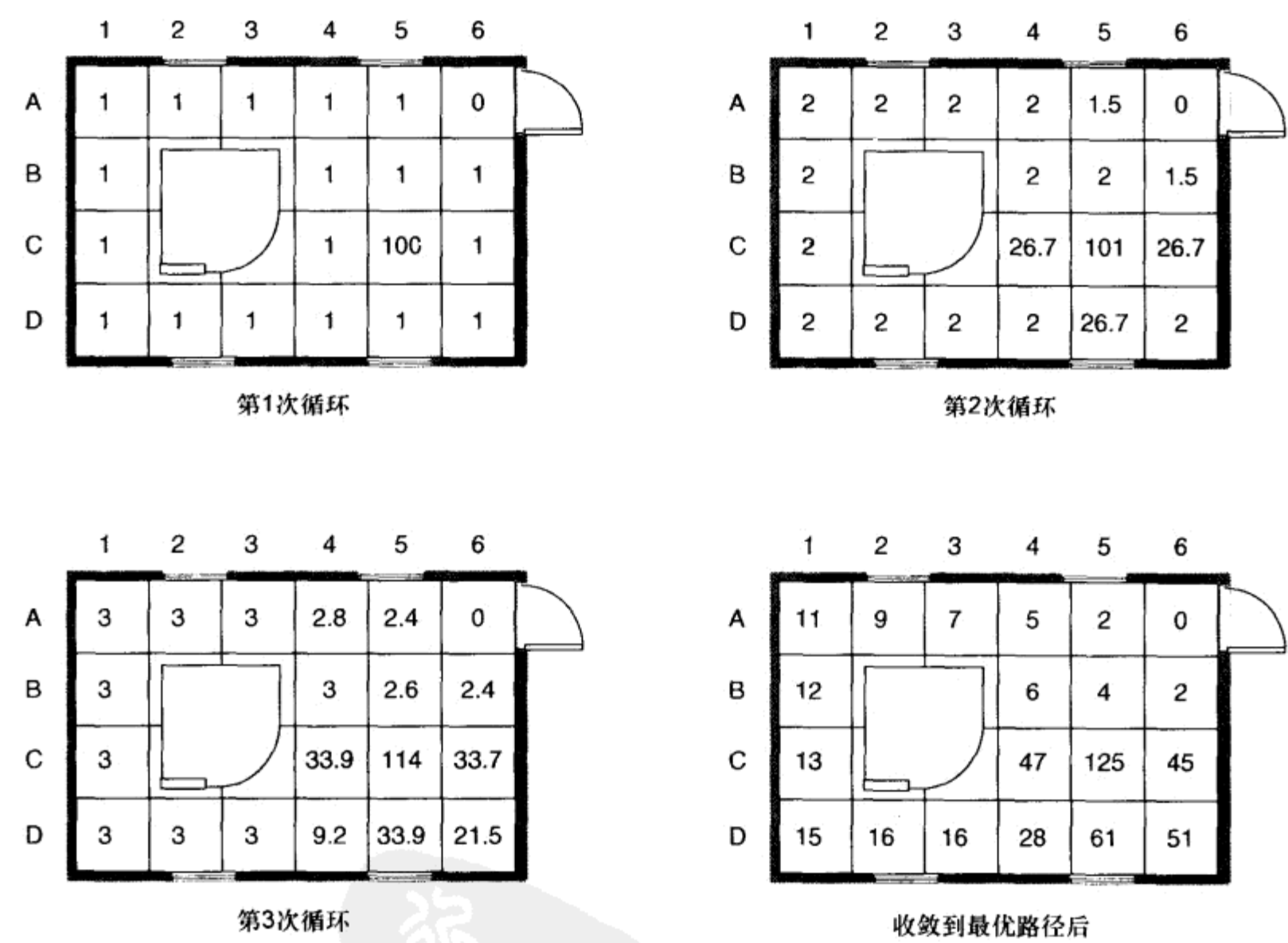


图 4.3.3 不同循环时的值表

在伪码第二步 c 中，我们用该格子的代价加上上步计算出的代价最小动作的值来作为该格子的新值。例如，图 4.3.3 周期一中，D5 有三个动作可选——向左走，向右走和向上走，其代价分别为 25.75、25.75 和 50.5。因此，格子 D5 的新值（图 4.3.3，周期二）就是 1（D5

原来的值) 加上 25.75。

对于伪码中第三步的一个问题是：我们到底需要多少个周期才够用？DP 算法的理论证明数组中的代价值会收敛（稳定），但是有可能会花很长时间。因为我们其实并不真正关心数组中的值到底是什么，我们关心的是是否得到了最优的路径。明确了这一点，我们就不需要那么多的周期数了。最理想的情况是，我们就让程序运行能得到最佳路径的最短周期，但是这个很难，我们事先并不知道这个周期数。所以，DP 算法是一种依赖于细节层次（LOD）的算法（有时也叫“随时（anytime）”算法）。它可以马上给出一个近似的方案，如果条件允许，它会继续算下去给出更好的方案。

最后，我们来考虑一下如何从格子代价值数组中获得最佳路径，假设最佳路径的结果保存在数组 A 中，伪码如下。

- (1) 用一个可能的动作来初始化结果数组 A。
- (2) 每个格子：
 - a. 对每个在该格子上可能的动作。
 - i. 计算出该动作的代价值；
 - b. 把有最小代价的动作保存到结果数组 A 中（如果几个动作代价值相同，取缺省的一个或随机取一个）。

显示了在头三个周期和收敛后的周期中算出的最优路径。

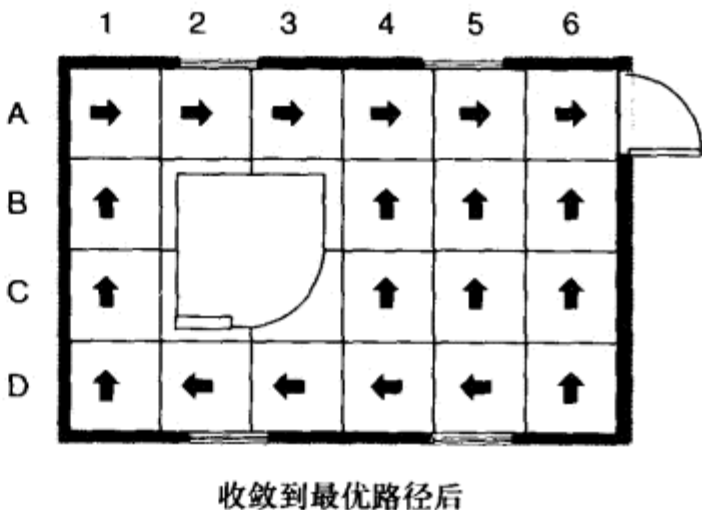
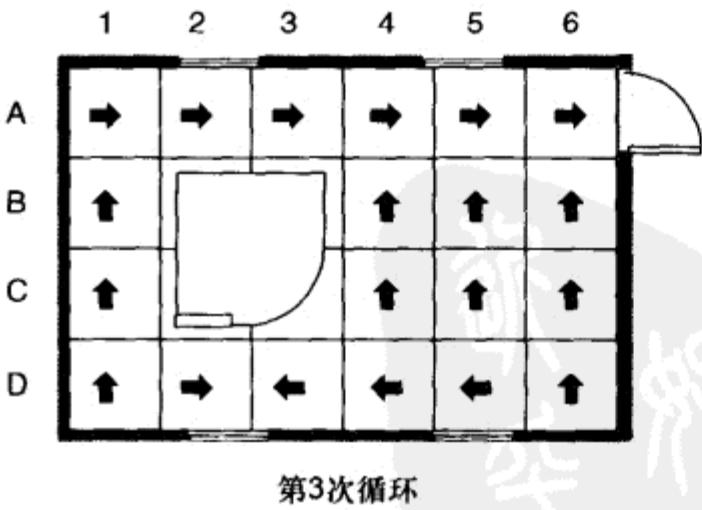
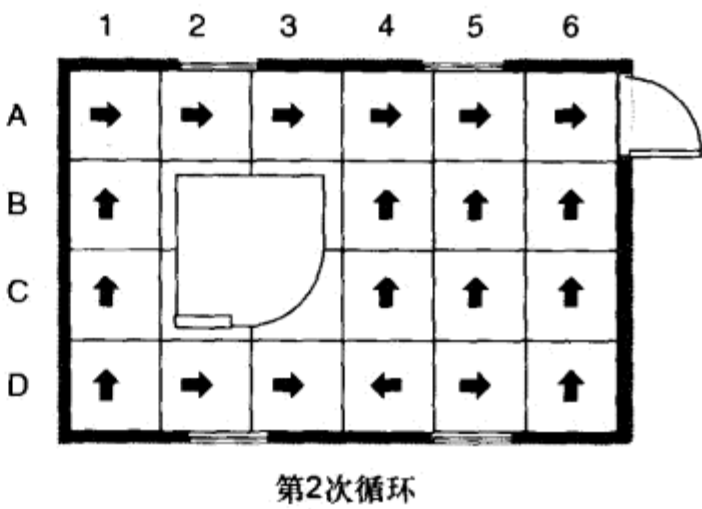
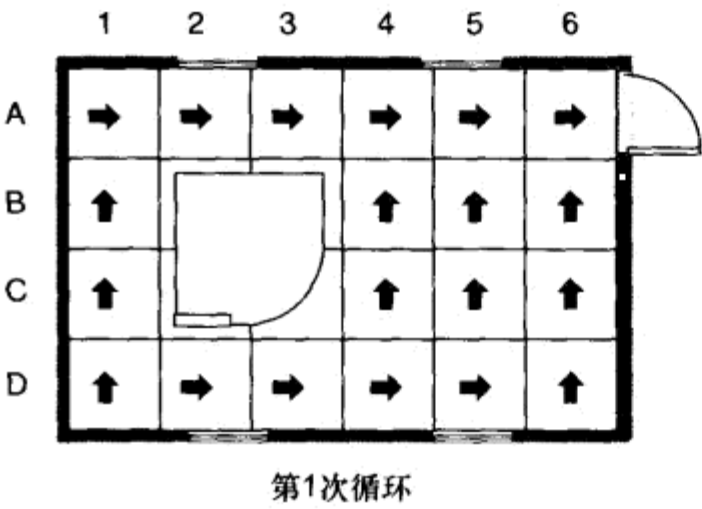


图 4.3.4 当前最好路径

4.3.3 代码

先定义各个行动概率表这个结构。

```
struct CAction {  
    float m_probStraight;  
    float m_probLeft;  
    float m_probRight;  
    float m_probBack;  
}
```

整个地图信息存在类 CMap 中，数组 m_walls 存着墙的位置信息，数组 m_costs 存着格子代价信息。

```
class CMap {  
    bool    m_walls[WIDTH][HEIGHT]; // 如果是墙则取“true”值  
    CAction m_actionProbs;  
    float   m_costs[WIDTH][HEIGHT];  
}
```

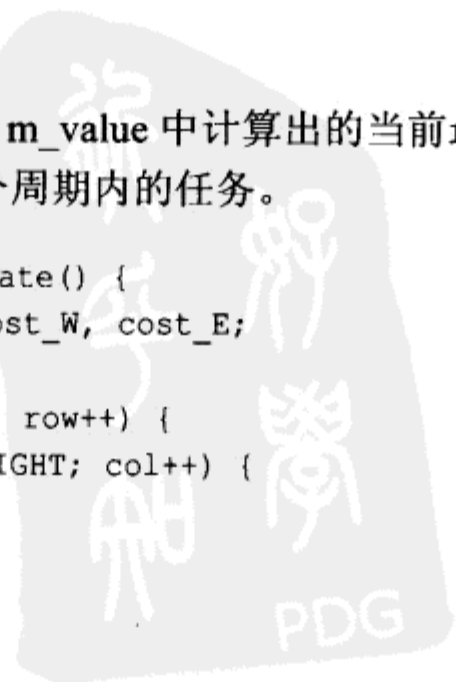
除了上述成员变量，CMap 还存着一些成员函数来操作地图数据，我们还定义了一个辅助类 CValue 来操作格子数据，这些类能有助于简化主算法的表述。在最终的代码中，这些被进一步优化了。

主类 CDynamicProgrammingAlg 存着算法相关的数据结构和算法本身。

```
class CDynamicProgrammingAlg {  
private:  
    CMap    m_map;  
    CValue  m_value;  
    char    m_bestAction[WIDTH][HEIGHT];  
    int     m_goalRow, m_goalCol;  
  
    float ComputeCostW(int row, int col);  
    float ComputeCostE(int row, int col);  
    float ComputeCostS(int row, int col);  
    float ComputeCostN(int row, int col);  
  
public:  
    void Iterate();  
}
```

数组 m_bestAction 保存着根据数组 m_value 中计算出的当前最佳动作。核心算法是在函数 Iterate() 中实现的，执行了算法每一个周期内的任务。

```
void CDynamicProgrammingAlg::Iterate() {  
    float cost, cost_N, cost_S, cost_W, cost_E;  
  
    for (int row = 0; row < WIDTH; row++) {  
        for (int col = 0; col < HEIGHT; col++) {
```



对于每个格子，我们都要计算可能发生动作的代价值，实现函数 `ComputeCostN()` 会在后便提供。

```
if (!m_map.isWallN(row, col))
    cost_N = ComputeCostN(row, col);
if (!m_map.isWallS(row, col))
    cost_S = ComputeCostS(row, col);
if (!m_map.isWallW(row, col))
    cost_W = ComputeCostW(row, col);
if (!m_map.isWallE(row, col))
    cost_E = ComputeCostE(row, col);
```

一旦得到了所有动作的代价值，我们要进行几个特殊的检查，比如看看是不是已到了目标格子或者是不是撞墙什么的了，如果没有什么特殊情况发生，我们就要计算最小代价的动作了。动作代价值被存在数组 `m_bestAction` 中，最小的代价值被用来更新格子值数组。

```
// 本格正是目标格
if ((row == m_goalRow) && (col == m_goalCol)) {
    cost = 0;
    m_bestAction[row][col] = GOAL;
}

// 本格是块墙壁
else if (m_map.isWall(row, col)) {
    cost = 0;
    m_bestAction[row][col] = WALL;
}

// 向北走代价最小
else if (cost_N <= _min(cost_S, cost_W, cost_E)) {
    cost = cost_N;
    m_bestAction[row][col] = NORTH;
}

// 向南走代价最小
else if (cost_S <= _min(cost_N, cost_E, cost_W)) {
    cost = cost_S;
    m_bestAction[row][col] = SOUTH;
}

// 向西走代价最小
else if (cost_W <= _min(cost_N, cost_S, cost_E)) {
    cost = cost_W;
    m_bestAction[row][col] = WEST;
}

// 向东走代价最小
else {
    cost = cost_E;
```



```

        m_bestAction[row][col] = EAST;
    }

    // 更新格子值数组
    m_value.set(row,col)=cost+m_map.getCost(row,col);
} } }

```

我们用向北走来演示如何计算动作代价值。

```

float CDynamicProgrammingAlg::ComputeCostN(int row, int col) {
    float prob_north = m_map.getProbStraight(row, col);
    float prob_east = m_map.getProbRight(row, col);
    float prob_west = m_map.getProbLeft(row, col);

```

打算向北走时，我们其实可能走到北、东和西三个方向。如果东方或西方有墙挡着的话，年轻人会回到原来的格子（走不动）。

```

float prob_N = m_map.getProbStraight(row, col);
float prob_E = m_map.getProbRight(row, col);
float prob_W = m_map.getProbLeft(row, col);

if (m_map.isWallE(row, col))
    prob_E = 0;

if (m_map.isWallW(row, col))
    prob_W = 0;

return prob_N * m_value.getValueNorthOf(row, col) +
       prob_W * m_value.getValueWestOf(row, col) +
       prob_E * m_value.getValueEastOf(row, col) +
       (1 - prob_N - prob_W - prob_E) *
       m_value.getValue(row, col);

```

4.3.4 优化

上边描述的 DP 算法实现在大地图上效率是不高的，我们可以用几个优化措施来改进。第一，我们可以先让算法在一个包含起点和终点的子图上运行，但是不包括大多数离它们都很远的部分。然后，如果某些原来没包括进子图的资源变得可以计算了，我们就扩大那个子图。上一个子图已经计算好的东西可以作为新图的初始信息，因为利用了以前的计算结果，无疑会省很多处理。

第二个大的优化来自于 DP 算法“精确度”的概念。也就是说，第一个周期结束后，算法就已经给出一些路径，如果算更多的周期，结果会更好。所以，计算分布在很多个周期上，非玩家角色可以随时拿到一个结果。

4.3.5 DP 算法的其他应用

现在，我们来看看 DP 算法在游戏中的其他应用。

假设现在有一个计算机巫师要找你斗法，要应用 DP 算法，我们要设置目标状态、动作、状态空间和代价。在这里，目标状态很简单，就是玩家倒下死掉。动作有三个：一是进攻，这样能有机会伤到玩家；二是防守，给自己疗伤；三是走到别的（更有利）的地方。状态必须要把所有关于决斗的东西都描述出来：还剩下多少条咒语，双方当前的点数，位置等等。我们可以把状态写成一个数组（剩下咒语，点数，巫师位置，玩家位置）。DP 的搜索状态空间是以上几个变量的所有组合，各个状态是用动作连接起来的，比如，念了一条疗伤咒语把状态 (a, b, c, d, e) 和 $(a - 1, b + 30, c, d, e)$ 以概率为 1 联系起来，这个状态的代价就是玩家所剩点数。

代价值数组的大小现在就等于状态空间的大小，DP 算法会像这样工作：计算出每一个状态上，每一个可用动作的代价值，选出其中最小的，然后更新代价值数组，当算法收敛时，我们就根据代价值数组输出最佳动作序列。

对于上边的那场斗法，为了更容易地建立搜索，我们用了高度概括的描述，我们也可以加入更多的细节，像咒语就可以分为魔法弹、催眠或盾牌什么的，还可以用更多的变量（比如角色的装备还有状态——清醒 / 昏睡 / 半梦半醒）来扩展状态空间，在算法实现上多花些精力或者多给些搜索时间，我们可以得到更复杂更有趣的策略。

4.3.6 结论

在这篇文章里，我们讨论了在随机地图找最佳路径的难点，我们同时讨论了 A* 算法应用在这种地图上的缺点，接着我们引入了擅长处理不确定因素的动态规划（DP）算法，并详细讲述了 DP 算法在随机地图上的应用，以及如何扩展到其他游戏场景中。

有多种基于基本 DP 算法的扩展，使它能应用到更大范围的游戏场景。游戏设计者更感兴趣的一类 DP 算法变种是加入了学习和采样能力的，这种改进使 DP 算法不再需要知道各个动作的概率甚至整个状态空间。算法所需要的数据是通过和自己或者玩家玩游戏的过程而自己生成的。实际上，这也是为什么基于 DP 算法的俄罗斯方块和西洋双陆棋计算机程序那么厉害的原因。这种 DP 扩展叫做反馈式学习（reinforcement learning）或者神经元动态规划（Neuro-Dynamic Programming），参见[Bertsekas96]和[Sutton98]，不过那已超出了本篇文章的范围。

4.3.7 参考文献

[Bertsekas96] Bertsekas, Dimitri, *Neuro-Dynamic Programming*, Athena Scientific, 1996.

[Bertsekas01] Bertsekas, Dimitri, *Dynamic Programming and Optimal Control: Second Edition*, Athena Scientific, 2001.

[Higgins02] Higgins, Dan, "Generic A* Pathfinding," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Matthews02] Matthews, James, "Basic A* Pathfinding Made Simple," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Sutton98] Sutton, Richard, *Reinforcement Learning: An Introduction*, MIT Press, 1998.

4.4 一个基于效用的面向对象决策架构

作者: John Hancock, LucasArts

E-mail: jhancock93@post.harvard.edu

译者: 肖昱

审校: 李劲松

游戏中的人工智能部分通常显得很臃肿, 充满了各种特殊规则和启发式的方法用来作决策。用启发式的方法并没有错, 好的启发式方法能节省很多珍贵的 CPU 资源; 但在游戏这个领域, 不是很漂亮的或者说不是最优 (但可能很有趣) 的结果也是可以接受的。除非 CPU 资源很紧缺, 否则一般游戏 AI 程序员不会去找最优解, 毕竟我们的目的是让游戏有趣, 如果一个 AI 对手十全十美, 那玩家可能得不到什么乐趣。

但是臃肿的架构是不可以接受的, 我们都看到过 (或写过) 满是一堆 if-else 规则纠缠在一起的决策逻辑块。如果只有几个 if-else 规则, 我们还可以分出它们谁是谁, 但要是非常多的条件规则放在一起, 这种 AI 逻辑就变得难读、难维护以及难扩展。一个蹩脚的架构会降低灵活性、代码的可维护性, 并最终会拖累整个产品。鉴于在开发的后期, 游戏设计通常要做出改变, 所以设计灵活的代码是非常重要的。

这篇文章介绍了一种基于效用 (utility) 的面向对象决策架构, 与那些固定写死的决策逻辑块相比, 这种架构更加灵活和可维护。文章中描述的主要准则都来自决策理论, 同时这些准则已经被应用到很多游戏当中, 比如即时战略游戏 *Star Trek: Armada* 中的武器选择 AI, 和 *Star Wars: Obi-Wan* 中的角色 AI 状态机框架。像传统的 AI 专家系统一样, 这种架构通常也把人的知识用启发式规则的形式编码, 用的时候再解码。

机器学习的方法可以用来自动做出决策, 自动适应设计的改变, 以及避免用显式臃肿的决策逻辑块, 但在游戏这个领域, 机器学习面临着巨大挑战: 培训一个 AI 系统是很困难也很耗时的 (因为评估 AI 性能需要花掉一个游戏的好几分钟或者几个小时); 而且, AI 系统的性能评估可能是不可靠的, 因为具体的性能评价指标 (比如输或者赢) 不仅仅是由 AI 系统自己决定的, 还要取决于对手 (AI 或者玩家) 的表现; 最后一点, 没有合适的方法来评价什么是 “有趣”。

如果你有幸获得足够多的训练数据 (很可能是从玩家身上), 你可以选择使用神经网络[Champanand02]、贝叶斯分类器[McCallum98]、决策树生成器[Quinlan93], 或者很多其他工具来帮你替换掉显式的决策逻辑块, 而

且它们统计出的或学到的知识能很轻松地和人知识结合起来，但是，这篇文章假设我们并没有训练数据来支持那些优化的决策算法。

4.4.1 决策树

有限自动机（FSM）架构在游戏的 AI 中应用广泛，决策树通常被用来决定状态之间的转换。例如：

```
if (IsInjured())
    RunAway();      // 优先级 1: 保护自己
else if (IsEnemyPresent())
    Attack();       // 优先级 2: 攻击敌人
else if (HeardNoise())
    Investigate();  // 优先级 3: 收集信息
else
    Patrol();       // 没有更合适的事可做
```

上边这样简单的决策树是有效的而且易读的，但通常的显式决策树是很脆弱的：随着可能的分支增多，可扩展性很差。如果一个游戏有 10 个相似但是功能稍有不同的 AI 个体，就可能要用 10 个像上边例子那样的显式决策树，或者一个有着很多分支（每个分支可能代表一种敌人）的大的决策树。用 10 个基本上相同的决策树会造成代码的冗余，并会在无意间增加那段代码中 bug 的数量。

但是，如果用一棵大的决策树，当需要加入第 11 个敌人类型或者前边没有的一个附加行为时，我们可能重写很多代码。总之，无论用哪一种方法，维护、扩展决策树都是费时而且没有保障的。所以，我们这里要实现的是一种隐式的决策树，它用面向对象的设计方法最终让维护更轻松、代码更灵活。

4.4.2 基于对象的更好的体系结构

状态模式[Gamma95]（软件工程设计模式中的一种）建议我们应该用对象而不是用函数的方式来实现 AI 的行为或状态，这样做的话会允许我们创建能容纳任意多的行为的隐式决策树而不用修改。

假设每一个 AI 角色都有一个容纳 AIState 对象的容器，我们给每个 AIState 对象一个 GetUtility() 函数来评估并返回它自己的重要性（用一个浮点数来表示），这样一个 AI 角色的 SelectState() 函数看起来可能是这样的：

```
AIState* CharacterAI::SelectState()
{
    AIState* pBest = NULL;
    float bestUtility = 0.0f;
    for (AIStateList::iterator j = States.begin();
        j != States.end(); ++j)
    {
        float utility = (*j)->GetUtility(this);
```

```
        if (utility > bestUtility)
        {
            pBest = *j;
            bestUtility = utility;
        }
    }
    return pBest;
}
```

注意，我们能够从一个角色中添加删除状态或者修改状态的功能而不需要丝毫改变 `SelectState()`，`AIState` 的子类总的来说是依赖类 `CharacterAI` 的，我们不需要把 `CharacterAI` 中的代码涉及具体的状态相关的细节，此外，如果一个角色没有自己特殊的状态或行为，我们就不需算它的效用值。

效用这个概念在决策理论中有着严格的定义，我们会在文章的后边仔细讨论它。现在，我们就把一个行为的效用值简单描述成它能多大程度完成高层目标，再用目标的重要性来加权。在一个射击游戏中，最高的目标就是活着并且杀死敌人，次要目标可以包括探测地图、获得东西和信息。如果按重要性把这些目标排序，我们也要按照同样的顺序来建立与各个目标相对应的状态的效用值。

如果一个状态和当前目标无关或者不能完成当前目标，那它的效用值就为 0。比如，如果当前没有敌人进攻，那进攻状态就是无关的，其效用值就为 0。按照重要性排序，建立前边所说的决策树，从逃跑（最高优先级行为），到攻击，再到侦察，最后到优先级最低的巡逻，建立一个效用值衡量体系并严格按照标准执行是非常重要的，特别是对那些缺乏具体的衡量方法的效用值。

为了实现和上边的决策树同样的行为，我们要定义行为的效用函数。

```
float RunAway::GetUtility(CharacterAI *AI) const {
    return AI->IsInjured() ? 0.9f : 0.0f;
}
float Attack::GetUtility(CharacterAI *AI) const {
    return AI->IsEnemyPresent() ? 0.6f : 0.0f;
}
float Investigate::GetUtility(CharacterAI *AI) const {
    return AI->HeardNoise() ? 0.3f : 0.0f;
}
float Patrol::GetUtility(CharacterAI *AI) const {
    return 0.1f;
}
```

这种解决方案看起来要比决策树复杂得多，但所带来的灵活性足以做出补偿。如此，我们就可以把具有不同动作组合的 AI 角色集成在一起，而不再需要为每个角色写一个决策树。当我们要实现一个新的状态时，必须根据它与其他已存在状态的重要性比例关系给出 `GetUtility()` 函数。

效用这个模糊的概念对构造一个更加灵活的有限状态机非常有用，但如果没有一个具体的方式来衡量或定义效用，写出紧凑的大规模基于效用的函数会很困难。为了更精确地定义效用，我们首先用一个武器选择系统为例子来解释一下期望值的概念。

4.4.3 期望值

决策者常常要面临在不确定的环境中做出决定的场景，更复杂的情形是，选择的动作产生的结果本身可能也带有不确定性。

在游戏中，可以做到创造出一个能知道所有过去和现在信息的 AI 角色，但这么多的信息需要很大代价来存储，而且如果有些信息本来是 AI 角色不应该知道的，那计算机其实是在作弊。AI 角色也许能预测一些未来的信息，但肯定不能很确定地预测玩家下一步的动作，这样，游戏中的 AI 角色必须要在不确定的条件下做出决策。

通常上，当一个人面临决策时，他会在可选方案中选择一个能产生最好结果的，但怎样定义“最好”结果呢？

在决策理论中，一个传统的定义“最好”的方法是最大期望值。一个动作的期望值是它的平均代价。数学上，一个动作的数学期望定义为，动作可能产生的所有后果乘上各自对应的概率值，最后的累加和。

$$E(V) = \sum_i p_i r_i$$

我们可以用这个理论来编出一个可以选择最佳武器配置的 FPS 角色。一种方法就是，计算出对于给定目标，各种武器对其伤害的数学期望值（因为我们选择武器的最终目的是要杀死敌人）。我们需要每种武器实现一个 `GetExpectedDamage()` 函数，来返回如果这个武器被选择后对敌人的伤害值。选择武器的决策函数和上边提到的 `SelectState()` 函数很类似——我们只需遍历各个武器然后找出有最大伤害期望的一个就行。

到现在为止，我们还没有讨论 `GetExpectedDamage()` 和 `GetUtility()` 这两个函数的参数，假设伤害值和距离有关系，我们可以给出：

```
virtual float GetExpectedDamage(float Distance) const;
```

我们还可以假设武器的伤害值和命中率都和目标距离有关，如果没击中目标，伤害值为 0，因此：

```
float Weapon::GetExpectedDamage(float Distance) const {  
    return HitChance(Distance) * Damage(Distance);  
}
```

每一个 `Weapon` 的子类都可以覆盖 `GetExpectedDamage()`、`HitChance()` 和 `Damage()` 这些函数来定义它自己的伤害期望值。

应变计划

一个好的程序员应该有应变计划。上边的 `GetExpectedDamage()` 函数对一些简单武器（比如手枪、步枪等等）的选择，只用距离这个参数来计算伤害值也许就足够了，但对于爆炸性武器，就不好用了。例如，要想算出一个手榴弹的伤害值，我们就要减去不想要的间接伤害值，这样的话，我们就需要手榴弹伤害范围内盟友的信息才能算出不想要伤害的大小和概率。在开发过程中给函数 `GetExpectedDamage()` 增加一个参数可能会影响到它的函数定义以及很多

其他已经在用它的地方，所以我们最好在一开始就定义好一个更灵活的可以应付变化的接口。

加入一个辅助结构 `WeaponContext`，能使我们的设计更具可维护性，这个辅助结构被用来存储所有常用的数据，我们的函数参数就是一个例子。

```
virtual float ExpectedDamage(const WeaponContext& context);  
virtual float Damage(const WeaponContext& context);  
virtual float HitChance(const WeaponContext& context);
```

使用一个像 `WeaponContext` 这样的结构，以后就可以随心所欲地增加函数的参数而不用担心修改已经存在的函数定义；通过提供一个更方便存储上下文信息（例子中就是计算多种武器的伤害期望值所需的信息）的地方，也可以提高效率。

我们把基于期望值的武器选择系统应用到 *Star Trek™: Armada*，一个即时战略游戏，有超过 40 种的不同武器[Activision00]可供选择。武器有不同的效果：有些会给同盟飞船造成伤害，有些能破坏防御系统，有些只能给人员造成伤害，还有一些能修补或保护同盟飞船。因为这么多不同的效果，不能简单定义一个期望值来作为伤害估计，而是要综合考虑对目标值的伤害、对友军的正面作用及负面伤害、还有很多其他影响来定义这个期望值，所以这样一个期望值从已知信息里很难轻松地计算出来，专家通常构造启发式的方法来解决这个问题。

面临着这么多的武器选择，不可能用一个显式的决策树来实现。用面向对象的结构来实现这样的系统，编程的轻松和灵活性并不是其带来的惟一好处：*Armada* 中的 AI 角色常常用我们从来没有为之编程过的致命武器组合让我们大吃一惊，这种效果是一种即兴的智能，对应变换的环境、武器切换的效用同时改变。

为了在你的系统里获得上边所描述的智能，你的期望值函数应该利用人类专家的知识还有你要评估对象的特性——这也是为什么 `GetUtility()` 和 `GetExpectedDamage()` 定义成虚函数的原因了。通常，并发的智能是一组非常简单的规则（存在各个分布的效用函数中）之间交互作用的结果。当然，在评估函数里（不管是任何理由喜欢还是不喜欢对象或者行为）只要不违反模块的独立性而且不耗费太多 CPU 资源，你也可以随意加入各种东西；但如果你发现在多个评估函数里有重复计算，那时就要把它提取出来并把结果放到辅助结构中去。

4.4.4 其他的决策准则

期望值并不是惟一的一个好决策准则，人们经常不是通过最大化期望值来做出决策的。

举个例子来说，假设有一种不寻常的彩票，如果你花 \$20 000 你有 50% 的可能性赢 \$50 000，买这个彩票的收益期望值是 $0.5(50\,000) + 0.5(0) = \$25\,000$ ，不买的期望值是 \$20 000（不花钱）。期望值理论说任何有理智的人都应该买这个彩票，因为这样的期望值最大，但是现实中很多人不会买，因为他们拒绝冒损失 \$20 000 的风险，这就是一人常常不根据期望值决策的例子。

最大化期望值虽然是完全合理（而且明智）的，但其他决策准则会给我们能力设计出更人性化的 AI 角色。比如，“最大最小”标准会先列出每个行动的最差结果，然后在其中选一个最好的（也就是说，寻找“最好”的最差结果），所以它也叫悲观（或规避风险）准则。

“最大最大”准则先列出各个动作的最好结果，然后再在其中选最好的那个（也就是说，

寻找“最好”的最好结果)[Winston91]。它是一种乐观准则。在效用理论中，我们会表述这两种准则，还有期望值准则及其他更多的准则。

效用理论

效用理论提供了一种方法来解释为什么面对同样的数据，不同的人会做出不同（但合理）的决策。最初，效用函数是对不同个体对不同抽彩条件的不同偏好的数学表述[Winston91]，其本质上是不同个体对风险的不同态度的数学表述。

冯·诺伊曼摩根斯坦效用理论（Von Neumann-Morgenstern Utility Theory）阐述了正常人会选择最大化期望效用的决策：

$$E(U) = \sum_i p_i u(r_i)$$

其中效用函数 $u(x)$ 在可能的收益上是递增的 ($u'(x) > 0$)，根据可微分的（并不是必须的） $u(x)$ 把决策者可以分为以下三种类型。

- (1) 风险规避型： $u(x)$ 是下凹的，即 $u''(x) < 0$ 。
- (2) 风险中立型： $u(x)$ 是线性的，即 $u''(x) = 0$ 。
- (3) 风险喜好型： $u(x)$ 是上凸的，即 $u''(x) > 0$ 。

给不同角色赋予惟一的效用函数会使他们具有与众不同（可信且前后一致）的个性。因为我们并不在乎作出个真人一样的个性，所以我们可以随意选择能达到期望效果的效用函数。例如，我们可以选择： $u(x) = x$ 作为一个风险中立型的个性； $u(x) = x^2$ 作为一个风险喜好型的个性；用 $u(x) = \sqrt{x}$ 作为一个风险规避型的个性。或者，用一个分段式线性函数或者反应曲线[Alexander02]来混合喜好风险的行为（买彩票）和躲避风险的行为（买保险，很多人的选择）。

以下函数定义基于最大化效用决策准则：

```
float Weapon::GetExpectedUtility(const WeaponContext& context) const
{
    return HitChance(Distance) *
           context.Owner->Utility(Damage(Distance));
}
```

现在，武器的期望效用值依赖于武器拥有者的 `GetUtility()` 函数，也就是说，取决于 AI 角色它自己（可以通过 `WeaponContext` 对象传入函数）。这套架构允许根据不同的角色或者个性产生不同的决策，而不用改任何决策逻辑块。

4.4.5 结论

在这篇文章中，我们介绍了一个面向对象的决策架构，以及包括期望效用在内的不同的决策准则。

生成概率估计也许是文中这种方法的一个重要缺陷，但我们不必非常逼真地构建一套模型，毕竟，我们不需要最优的结果。我们需要的概率值有时在游戏的逻辑中已经给出了，有时，这些概率可以手工估计或者用贝叶斯推理系统分析玩游戏过程中产生的数据而获得。只

要不同的 AI 角色有不同的风险喜好类型，不同的效用函数就会产生不同的决策，这样就产生了不同个性的角色。

不管你准不准备把效用理论（也许你喜欢别的决策准则）应用到 AI 角色中，文中描述的这种分布式的架构会给你的程序带来巨大的好处：灵活性、可维护性以及即兴发挥性。但如果你能用效用理论的话，你会用很小的代价来人性化你的 AI 角色决策过程。

4.4.6 参考文献

[Alexander02] Alexander, Bob, “The Beauty of Response Curves,” *AI Game Programming Wisdom*, Charles River Media, 2002.

[Activision00] *Star Trek: Armada*, Activision, Inc., 2000.

[Champanand02] Champanand, Alex, “The Dark Art of Neural Networks,” *AI Game Programming Wisdom*, Charles River Media, 2002.

[Gamma95] Gamma, Erich, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[McCallum98] McCallum, A., Nigam, K., “A Comparison of Event Models for Naive Bayes Text Classification,” *AAAI-98 Workshop on Learning for Text Categorization*, 1998.

[Quinlan93] Quinlan, J. R., *C4.5: Programs for Machine Learning*, Morgan Kaufmann, 1993.

[Winston91] Winston, Wayne, *Operations Research: Applications and Algorithms*, Second Edition, PWS-Kent Publishing Co., 1991.



4.5 一个分布式推理投票架构

作者: John Hancock, LucasArts

E-mail: jhancock93@post.harvard.edu

译者: 肖罡

审校: 李劲松

多数基于行为的 AI 架构都不能同时监控多个行为模块: 在任意时候, AI 系统只能执行当前激活的行为或者状态而不去管其他的。所以这些系统不能在行为之间做出协调或者不能同时满足多个行为的目的。除非我们专门为某个特定行设计和实现它与其他行为的通信(有时会给 AI 架构带来不用忽视的代码依靠性问题), 否则, 非焦点行为的信息都要被浪费掉。

这篇文章讨论了一个 AI 架构, 它由多个独立的推理模块组成(以后简称顾问), 还有一个仲裁者来综合各个顾问的意见, 这种基于投票的架构有着易于使用、易于实现以及易于维护的优点。

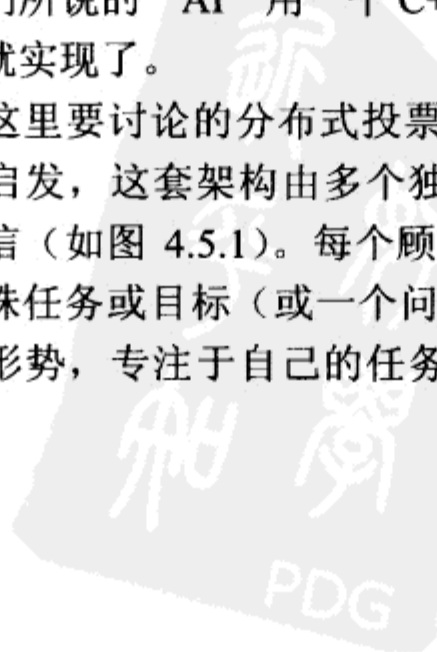
我们定义一个“投票空间”来作为顾问们可选意见的集合, 开发者可以分别实现不同的顾问(只要顾问输出的选票落在投票空间内)。

这篇文章里描述的架构可以融合异构的模块, 比如, 低层的模块以 30Hz 的频率运行, 高层模块(如寻路)以较低的频率运行, 同时还维持对反应式行为至关重要的高频输出。异步的友好合作对游戏中的 CPU 负载平衡是非常有用的, 尤其是当 CPU 资源非常紧张的时候。

4.5.1 分布式推理

分布式推理系统把认知的过程和任务分摊在多个系统或地方, 与之相对应的集中式架构使用单一的模块来负责认知和规划(虽然可能会用到多个资源以及其他模块来构造整体模型)。游戏中的 AI 系统通常是集中式的: 大多数我们所说的“AI”用一个 C++ 类来作为行为的大脑, 或者直接在行为的类中就实现了。

我们这里要讨论的分布式投票架构得到了 Rosenblatt [Rosenblatt97] 中成果的启发, 这套架构由多个独立的顾问组成, 它们都要与一个中央仲裁者通信(如图 4.5.1)。每个顾问都是个推理模块(或行为), 负责执行某个特殊任务或目标(或一个问题的一个方面), 它们(顾问)能够独立地评估形势, 专注于自己的任务, 并且能计算出它们每个可能行动的效用值。



顾问之间是独立的，也就是说，它们的决策不依赖与其他顾问的结果，同时，顾问之间也没有通信联系。这种相互独立性允许你在运行时添加、删除或者屏蔽某个顾问而不会影响到仲裁者和其他顾问。每个顾问都可以用任何可行的技术来实现，比如，第一个可以用贝叶斯推理系统，第二个可以用启发式知识，第三个可以用神经网络来投票。这种灵活性就是分布式结构相对于集中式的优势。

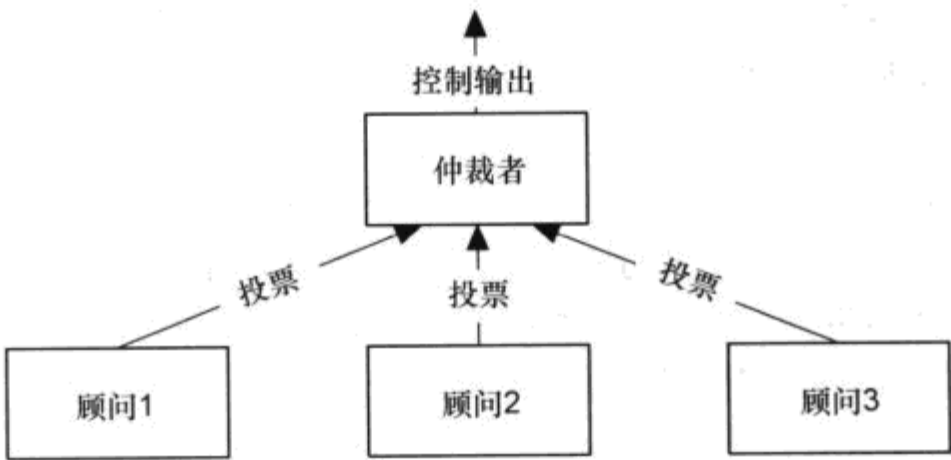


图 4.5.1 仲裁者综合多个活跃顾问所发出的投票，从中选出最好的一个，然后生成控制输出

1. 执行综合意见的错误方法

让我们先来研究一个移动机器人到目的地的问题，机器人要避开在寻路过程中地图上的静态障碍物，同时也要躲开动态的障碍物（比如车辆）。我们假设有一个顾问引导机器人沿着规划的路径走到目的地，还有一个顾问负责避开障碍物。一个天真的解决方案是根据每个顾问给出的方向矢量，取它们的平均值来作为最终行走方向，图 4.5.2 演示了用这种算法产生的一个灾难性结果。

实际上，上边就是“Potential Field”算法所做的，它平均了所有代表着排斥或吸引的矢量而产生一个综合向量，在任何时候（就像图 4.5.2 中那种情况），即使有更好的选择，用

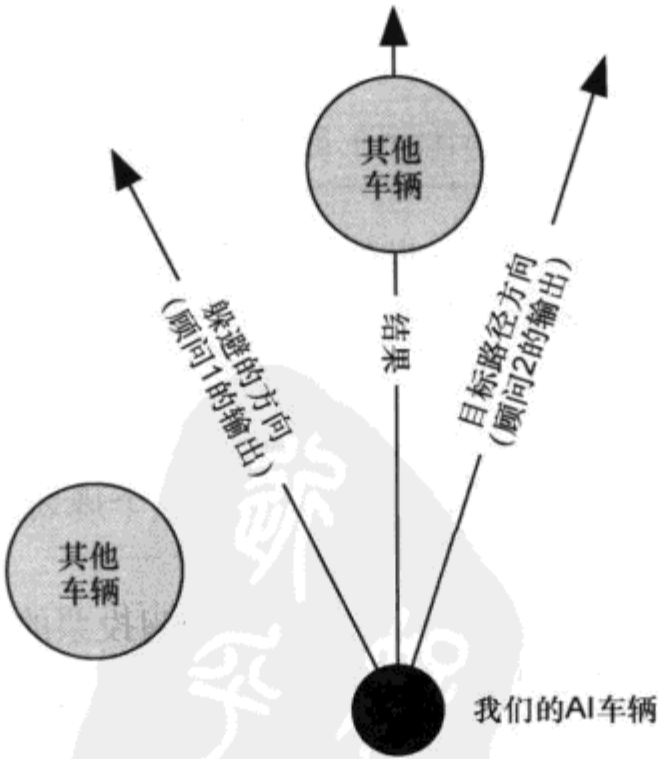


图 4.5.2 从各个顾问（一个障碍物规避顾问，一个目标路径引导顾问）得到的平均矢量可能会产生一个比任何一个单独顾问的建议都差的决策。在这个例子中，计算出的方向反而会使机器人撞到障碍物

“potential field”算法都有可能导致机器人撞到障碍物；另外，“矢量和”可能会产生使 AI 系统陷入困境的“局部最小”。假设 AI 系统没有陷入“局部最小”的陷阱，因为当机器人靠近时，障碍物的排斥力会变得更强，所以相撞的概率还时很小的。但是，我们能够做得更好。

与上边所述综合各个顾问决策的方法不同，很多系统就简单地把顾问决策按重要性排序。这样一个系统把控制权在一个时刻只交给一个处理模块（或者根据有限状态机中状态转换逻辑，或者根据像 Brooks 中“包含架构”[Brooks91]那样规定好优先级）。在上边的寻路例子里，当车辆快要撞到障碍物时，危险规避模块就接管了车辆的方向控制权，而不理会目的地引导模块的建议；一旦脱离的危险，目的地引导（寻路）模块就要接管控制权。当目标是排他的时候，用这种优先级排序的方法可以解决问题，但如果要有多个目标同时需要满足的话，那种方法就不好用了。在我们的分布式架构里，多个行为模块会通过仲裁者同时参与控制：很多时候，我们可以躲避开动态障碍物，同时却不用改变太多已经规划好的路径；还有很多时候，偏离规划好的路径需要我们重新计算路径。

2. 通过投票来综合意见

如果我们重新设计上边的分布式系统，让每个顾问一次对多个行为（而不只是一个）投票，那么仲裁者就会有更多的信息来处理，也就通过综合意见得到更好的决策。当一轮顾问投票中产生一个最优动作时，知道这个最好动作比其他动作到底好多少是非常重要的，因为最好的模块间的折衷方案可能会包括那些动作。

顾问通过投票（在投票空间上）与仲裁者通信，一个正面的选票表示了对某个动作的喜好，反之，负面的选票就说明不赞成，每张选票还有个程度值来表示喜好（不喜好）的程度，仲裁者从所有顾问那里收集选票，从而选择最佳的动作，然后输出给 AI 控制者。对顾问的惟一要求就是他们能和仲裁者建立起一种可理解的通信格式，并且选票的程度值要设计合适。通常，每个顾问的输出应该是在同一个投票空间内，而且选票的程度值被规范化在 $-1 \sim 1$ 之间。如果仲裁者能很好的理解溶合多种通信格式，使用统一的投票空间（或程度值）并不是必需的。

4.5.2 操纵仲裁者（Steering Arbiter）范例

出于演示的目的，我们现在假设正在设计一个 AI 系统，它能在一个车辆战斗游戏中的粗犷地形上自动驾驶车辆。具体来说，我们希望能设计出一个能在任何时刻为车辆选择转弯曲率的系统。理想的这样一个 AI 系统能同时做下面一个或所有的事情：躲开敌人火力，避开障碍物，消灭敌人车辆（通过撞击或者把敌人赶到自己的射程内），考虑车辆的动力学限制（避免翻车），以及向目标驾驶（如果希望的话可以通过一个规划好的路径）。这类问题非常适合我们的分布式架构。

要设计这样一个架构，我们第一步要做的是选择顾问投票的表示方法，让选票的表示方法和系统的最终输出方式相一致不是必需的，但力争让仲裁者做到领域独立或者不必知道专业知识（相反，各个顾问不是领域独立的）。为此，我们选择最佳的转弯弧度 ρ （也叫做曲率，定义为转弯半径的倒数），每一个顾问都要为定义域在 $(-1/R_{\min}, 1/R_{\min})$ 上的几个离散转弯弧度值投票，这里的 R_{\min} 是车辆的最小转弯半径（如图 4.5.3）。候选离散弧度数量的选择要平

衡考虑效率和给顾问足够多的选择，我们不必限定最终的系统输出是这些离散值中的一个，在后边我们示范如果根据离散数目的弧度值评价产生模拟值的输出。

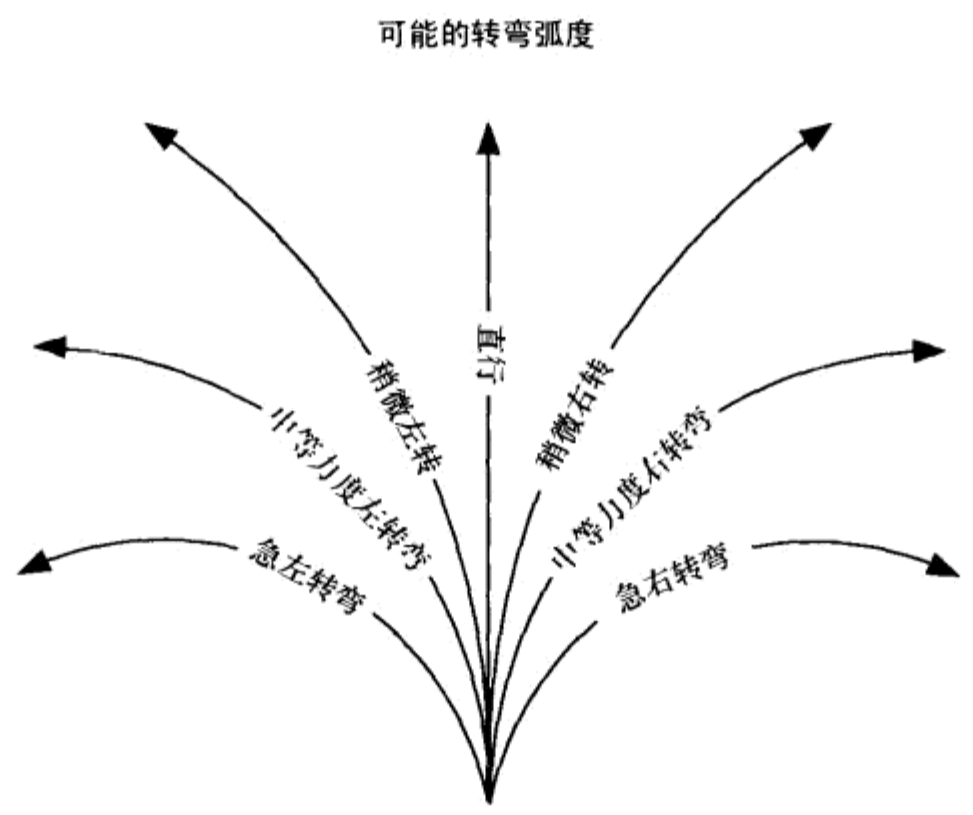


图 4.5.3 每个顾问必须要计算各个候选离散转弯弧度的效用值

我们接着针对下面各个目标创建对应的顾问：

- **避免翻车：**这个顾问会对任何会导致翻车的候选转弯弧度投反对票。低速的时候，所有的转弯弧度都是合理的，但在高速时，顾问会否决那些极大的弧度。
- **躲避障碍物：**这个障碍物躲避顾问（根据每个转弯弧度所需的最短距离以及沿着弧度到障碍物的距离）对那些有撞上障碍物危险的转弯弧度投否决票。与越近的障碍物有交叉的弧度，得到的否决程度值越高；相反，对于刚好可以擦身而过的近距离障碍物，或与稍微远一些的障碍物会相撞的转弯弧度（注：顾问只需考虑一定范围内的局部障碍物，远处的障碍物一般不会影响决策），顾问会投一个比较弱的反对票。
- **避免敌人火力：**这个顾问会对能与当前敌人的火力相交的转弯弧度进行否决，否决的程度取决于停留在敌人火力区的时间长短。
- **使自己的武器转向敌人：**这个顾问会对能使 AI 车辆很快向敌人车辆射击的转弯弧度投赞成票。
- **到达目的地：**这个顾问倾向于能让车辆尽快到达目的地的转弯方向。例如：如果车辆能以很快的频率更新转弯弧度的话，系统在这个顾问的影响下会输出车辆与目的地之间方向、车辆当前方向这两个矢量的点积。

最终的架构如图 4.5.4。

1. 仲裁者的设计

最后，我们必须决定仲裁者如何从它的顾问的输入中选择最好的转弯方向。比如有个系统有两个顾问来提供转弯弧度的意见，仲裁者有两种方法来综合顾问的意见（如图 4.5.5）。

注意，顾问的投票空间平滑覆盖了整个转弯弧度。

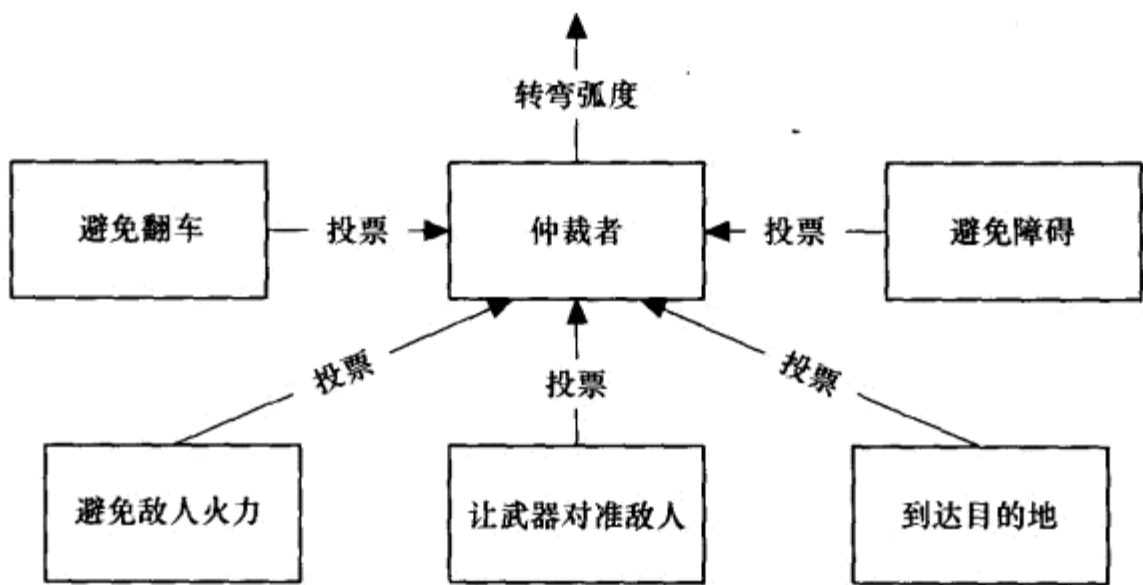


图 4.5.4 在车辆战斗游戏中，仲裁者为 AI 车辆选择转弯弧度的系统

通常来说，如果仲裁者希望能得到模拟输出（就像我们这个转弯仲裁者），那顾问的投票应该平滑地覆盖整个输入空间。平滑的投票分布（就是相近的输入会产生相近的投票值）会促使仲裁者做出妥协的决定来更好地真实表示模拟输出。我们可以或者用连续函数来产生顾问的投票，或者用高斯类的平滑函数来处理顾问的投票——也就是说，低通过滤投票 [Rosenblatt97]。

图 4.5.5 中第一个仲裁者会选择具有最大的两个顾问投票值和的行为，使系统选择稍微向右的一个转弯；第二个仲裁者叫“最大最小”（maximin），也就是说，会在两个顾问投票的最小值之间选择最大的一个，这样，系统会照直向前开，maximin 仲裁者是风险规避型的。

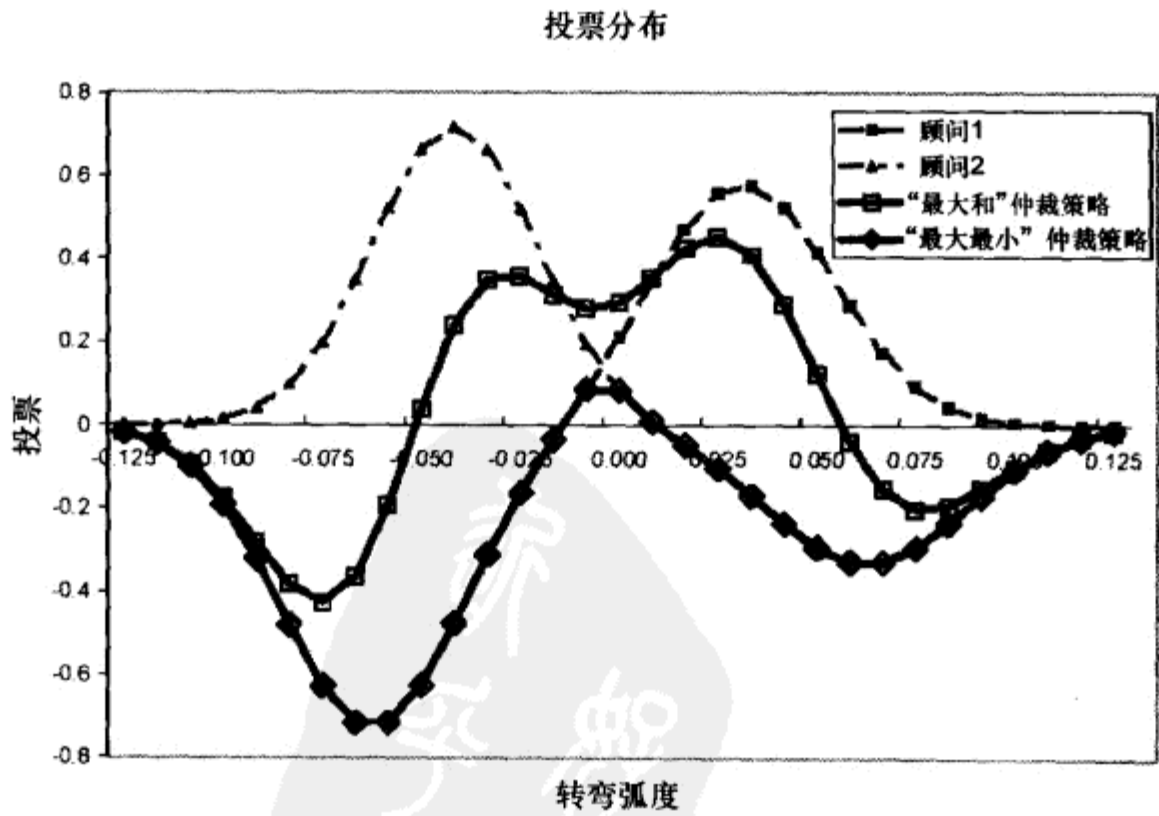


图 4.5.5 两个顾问、两种不同的仲裁策略的投票分布。第一种仲裁策略“最大和”，选择两个顾问投票值之和中的最大值。

第二种仲裁策略，“最大最小”，选择两个顾问投票值中最小那个，再在其中选取最大值输出

采用选择加权投票和中最大值策略的仲裁者是很高效的，用这种方法，与安全相关最重要的顾问会得到最高的权重，允许顾问有否决权的仲裁策略同样具有风险规避类型仲裁者的优点，我们可以通过把反对票的值提高几个数量级（相对于正常）的方法来实现否决。

2. 从离散投票值中产生模拟输出

把仲裁者的输出限定于离散的弧度值是我们转向仲裁者的一个重大缺陷，如果通过提高离散值数量的方法来增大输出结果的选择，代价可能会很高，幸运的是，我们的输出值可以不局限于在输入的离散值中取值，我们可以通过二次方插值的方法计算出模拟输出。

因为 3 点决定了一个二次方程，我们可以通过 3 个点（产生最大 y 值的 ρ_i ，以及与 ρ_i 相邻的 ρ_{i-1} 和 ρ_{i+1} ）来计算出二次方程的 3 个参数值 a 、 b 和 c 。

$$y = a\rho^2 + b\rho + c$$

接着，最好的弧度是像下面这样的：

$$\frac{\partial y}{\partial \rho} = 0 \Rightarrow \rho = \frac{-b}{2a}$$

这种方法能够保证输出结果能落在最好值的产生区间 (ρ_{i-1}, ρ_{i+1}) 中。

4.5.3 选择投票空间

对于一个要执行具体行动的 AI 系统，命令最终必须以控制指令的形式输出，如果是一个机器人实体，输出也许就是一套电流或电压。当有附加的抽象层提供更高级别的控制时，我们必须最终将这些高级控制转换到动作具体执行者的命令空间。在游戏设计中，程序员可以更加灵活地选择控制输出方式：对于一个车辆，如果需要，我们可以直接控制加速和转弯，所以不必构造发动机、方向连接系统以及很多其他东西的模型。在上边的例子中，我们用转弯弧度 ρ （转弯半径的倒数）来作为投票和控制空间。

对于大多数系统，投票空间可能会有多个选择。首先并且最重要的一个原则是，要能使顾问进行方便而有效的推理。如果投票空间和控制空间（决策输出）相同，那么就不需要有命令转换的过程。仲裁者通常要做到与领域无关或者没有专业知识，但因为只有一个仲裁者，如果需要，让仲裁者负责将投票空间转换到控制空间也是可以接受的。

为了演示投票空间不同于控制空间的情形，我们回到驾驶的例子，假设车辆顾问们根据离散的地图位置，而不是转弯弧度来投票。基于地图的投票空间也许会简化障碍物探测顾问的工作（因为它可以只是简单对地图上含有障碍物的格子投出反对票）；另一方面，会加重避免翻车顾问的负担（因为通过转弯半径更容易计算动能值）。当仲裁者处理与地图相关的选票并由此决定最好的转弯弧度时（假设弧度依旧是作为控制输出），基于地图的投票空间需要这个仲裁者具有相关领域知识。

但基于地图的方法有利于对的顾问封装（因为旧地图可以转换到新坐标上而不需修改顾问），如要了解更多关于基于地图的投票在车辆转弯问题上的应用，请参阅[Rosenblatt97]。

最后很重要的一点是，选择的投票空间要能给仲裁者提供足够的信息以便其做出明智的决定。当希望的仲裁输出是连续（实）值时，要想得到最佳的结果也许就需要投票空间的解析度足够高到能够构造出顾问的连续投票函数。Nyquist 样本理论指出，要想重新构造出

信号函数，采样频率必须是最大信号函数频率的两倍。实践中，最优的结果通常是不必要的，所以低频的采样解析度也可能是够用的。

即时战略游戏中的基地建造仲裁者

为了更深入地说明基于地图的投票系统，我们现在考虑一个即时战略游戏中的 AI 问题。

在即时战略游戏中，一个常见 AI 问题是基地的建造过程。对于一个目的是要征服敌人计算机玩家来说，它们需要利用可获得的资源和土地来扩展它们的帝国，从而支持和扩建军队。AI 玩家必须能够构建建筑物并且偶尔需要建新的基地，因为建筑物通常是不可移动的，建设之前的选址是非常重要的。

很多因素可以影响到新基地或新建筑物的选址，我们设计一个仲裁者来选择新建筑物的地址（假设 AI 系统已经选择好了一个建筑物的类型）。

控制空间要输出建筑物在地图上的位置，投票空间选择为可选位置的集合，下面是我们可能要创建的顾问。

- **自由空间：**否决不合法的位置，倾向于那些不阻塞基地交通的位置。
- **方便获取资源：**如果建筑物是用来获取资源的（比如 *Warcraft* 中的木头加工厂），可以用反应曲线等方法选择靠近相关资源的位置。
- **躲避敌人：**赞成票投给远离敌人基地的位置，相反投反对票。
- **需要保护：**如果一个重要建筑物没有自卫能力，对靠近基地或者防御建筑的位置投赞成票。
- **提供防护：**如果是像塔楼之类的防御性建筑，反对票投给该地方已经有了其他类似建筑，赞成票投给附近有重要战略价值建筑物的地址。

上面描述的任何一个顾问，或者仲裁者自己，都可以作为一个“影响图”来实现。并不是所有的顾问都和所有建筑物的选址问题相关，但我们的架构中允许我们很容易根据 AI 玩家要创建建筑物的类型来开 / 关各个顾问，并且，不同建筑物类型也可以自己给顾问它们特有的一些信息。

一个具有否决能力，基于加权求和的仲裁者可以给我们提供最好的结果，不同的权重可以赋予不同的 AI 玩家以不同的个性。对那些很重要的建筑物，我们可以考虑提高与安全相关的顾问的权重（我们想最小化丢失这些建筑物的损失）。

要找出分配给每个顾问的最好权值，也许要涉及到合理的猜想和反复的试验，如果不喜欢反复试验，遗传算法或者其他机器学习算法可以学习这些仲裁者要用到的权值并且可以自动调节顾问模块内部的参数值[Baluja97]。但是，因为最优的结果对“有趣”的结果来说不是必须的，为了得到最大的娱乐效果，某些手段是不可避免的。

4.5.4 结论

这篇文章里，我们介绍了一个架构，允许多个行为共同（同时并且部分）控制一个 AI 系统。这种投票的架构相对于基于状态或者优先级的架构有着明显的优势，能够综合各方面意见或一次完成多个任务。而且分布式的架构更加灵活，更容易维护，在未知环境中更加可靠。

因为投票架构要产生突发的行为,它也许不适用于那种需要集中规划来求最正确或最优结果的情况。相对于集中式架构,分布式架构产生的输出难以预测,但也正是这一点,使分布式架构更健壮而且可以适应未知的环境。幸运的是,这种架构对绝大多数游戏都是非常适用的,因为游戏设计中灵活性是非常重要的,而最优性很少是必要的,同时高度动态的游戏环境也使规划变得很难。

4.5.5 参考文献

[Baluja97] Baluja, S., R. Sukthankar, R., and J. Hancock, "Prototyping Intelligent Vehicle Modules Using Evolutionary Algorithms," *Evolutionary Algorithms in Engineering Applications*, Springer-Verlag, 1997.

[Brooks91] Brooks, R.A., "Intelligence without Representation," *Artificial Intelligence* 47 (1991), pp. 139–159.

[Rosenblatt97] Rosenblatt, J., "DAMN: A Distributed Architecture for Mobile Navigation," *Journal of Experimental and Theoretical Artificial Intelligence*, Vol. 9, No. 2 / 3, 1997, pp. 339–360.

[Sukthankar97] Sukthankar, R., "Situation Awareness for Tactical Driving," CMU Technical Report, CMU-RI-TR-97-08.

[Tozour01] Tozour, P., "Influence Mapping," *Game Programming Gems 2*, Charles River Media, 2001.



4.6 吸引子和排斥子

作者: John M. Olsen, Microsoft

E-mail: infix@xmission.com

译者: 沙鹰

审校: 李劲松

哪些东西应当接近, 哪些又应与其保持距离, 若你的由 AI 控制的角色能够明白这点, 将会对模拟真实的行为大有帮助。许多任务, 譬如行走穿过人群、在跑道上赛车和空间飞行这样的任务, 都包含接近某些物体, 同时避开另外一些物体的要求。吸引子 (attractor) 和排斥子 (repulsor) 有很多用途, 包括模拟群体行为 (flocking behavior)、避免赛车互相碰撞、在 2D 和 3D 环境中追踪对手等。我们可通过引力曲线 (attraction curve, 即确定物体之间的吸引与排斥的程度的函数) 来影响我们的 AI 角色的运动。我们也可以将数个简单的曲线组合成更复杂的合成曲线, 从而实现有趣的突现行为 (emergent behavior)。

该技术最为有用的场合是用来增强某个已有的移动系统或寻路系统的作用。吸引子和排斥子就其本身而言, 并不能作为你的主要寻路系统来使用, 这是因为无法指挥 AI 如何在复杂的障碍物附近导航。作为一种移动技术, 吸引子和排斥子能够使移动对象对环境做出有趣的反应。但是若没有底层的寻路系统, AI 就会在遇到例如局部极小值 (local minima) 这样的问题时陷入困境。下面我们首先介绍系统的机制, 然后介绍 AI 控制的不同层次之间的交互。

4.6.1 合力

为了使物体运动, 我们需要一种合成了作用于该物体上的吸引力与排斥力的综合效果。为此, 我们将某个截止距离 (cutoff distance) 范围内的全部吸引力与排斥力进行累加。然后根据简单的牛顿物理学公式 4.6.1, 将合力换算成加速度。加速度即合力除以物体的质量。

$$a = \frac{\sum f}{m} \quad (4.6.1)$$

对于本文中探讨的, 采取一种简单的欧拉积分的近似算法就可以满足要求了, 如公式 4.6.2 和 4.6.3 所示。逐帧地, 使加速度作用在原始速度上得出一个新的速度, 然后根据这个新的速度以及原始位置找出当前帧的新位置。这种近似方法效果尚可, 条件是时间间隔相对于速度来说要足够小。

$$v_f=v_0+a\times dt \tag{4.6.2}$$

$$p_f=p_0+v_f \tag{4.6.3}$$

有时我们需要使用更严格的方法，以避免振动（oscillation）和其他可能因较大的时间间隔或在力系统中增加阻尼而产生的误差等问题。当必要时，我们可以采用更精确的物理模拟方法，例如泰勒定理 [Lander99]。

4.6.2 引力曲线

引力曲线是确定物体相互作用力与它们之间距离的关系的函数。任何满足“对于任意距离值都有且仅有一个力值”的函数都可以用来描述引力曲线。设 y 是作用力的大小，以纵轴表示； x 是施力与受力的对象之间的距离，以横轴表示；即可认为 y 是 x 的函数。曲线不一定非得连续不可，但连续的曲线对于产生平滑的结果有着显著优势。图 4.6.1 和 4.6.2 是吸引曲线的例子。

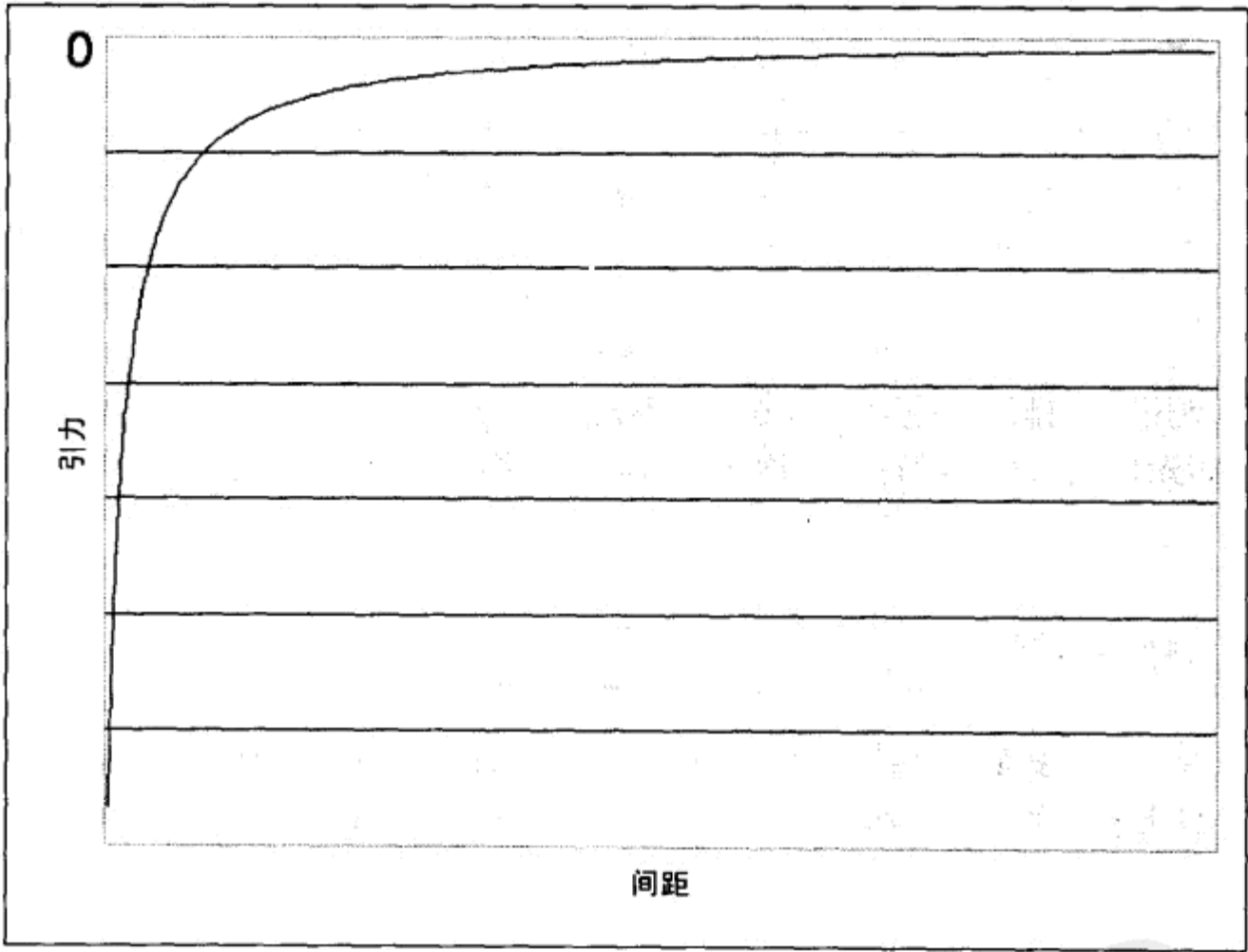


图 4.6.1 排斥曲线 $y=-x^{-1}$ 的图，越接近排斥力就越大

注意这两张图并不是完整的函数图像， x 的负值部分被去掉了，因为 x 轴代表物体之间的距离而非单纯代表位置。我们在此忽略负的距离值，因其通常无用。

我们必须特别注意避免使用那些在某处趋向于无穷的曲线。图 4.6.1 中所示的曲线在 $x=0$ 时趋向于负无穷，这就意味着我们需要进行额外的测试，以避免在物体上施加无穷或特别巨大的力。为了避免出现问题，一种简单的方法是将曲线整体稍稍向左平移，这样在 0 处 y 值也是有限的，如公式 4.6.4 所示。

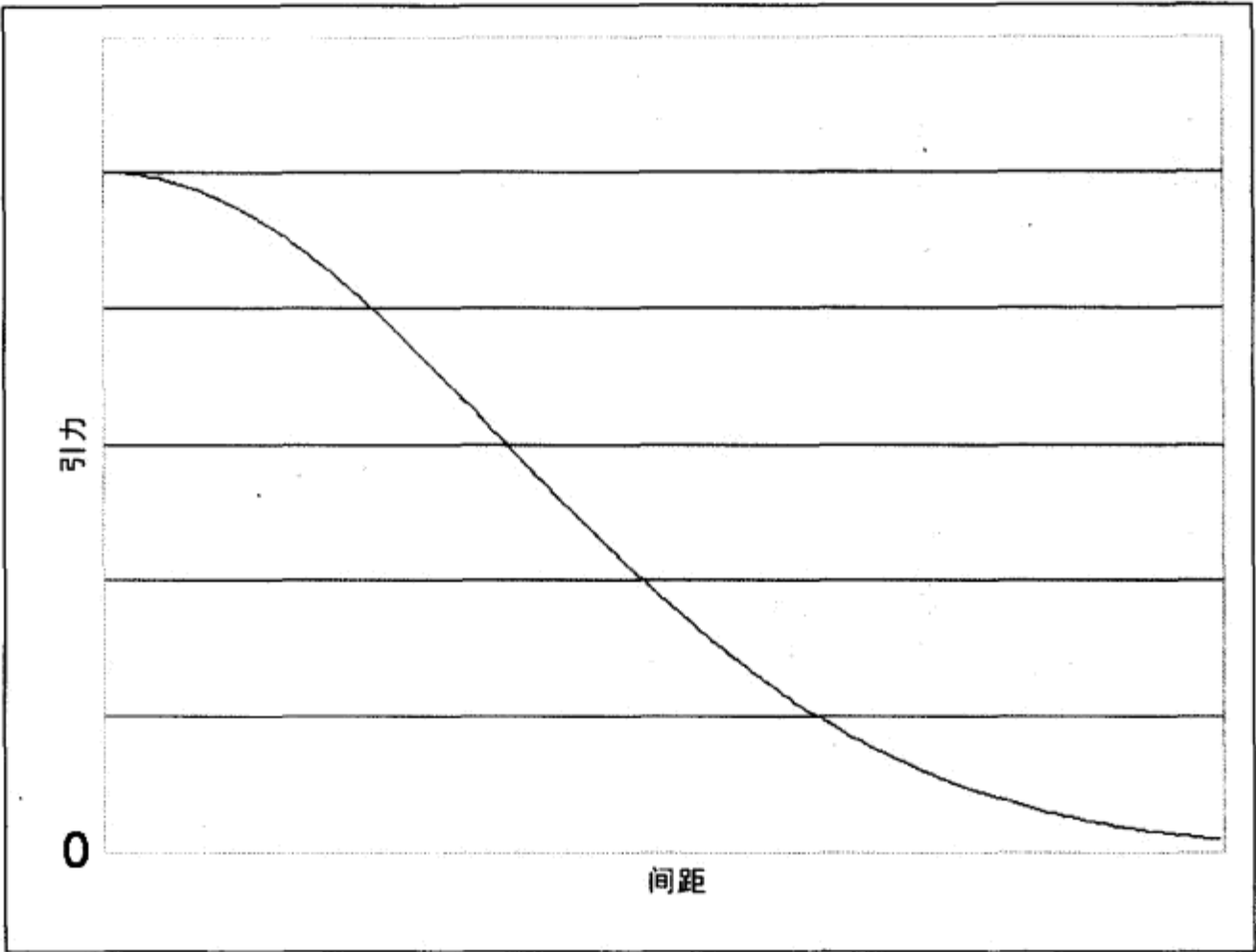


图 4.6.2 吸引子 e^{-x^2} 。铃形曲线上的 y 值为正数，越靠近吸引力就越大

$$y = \left(\frac{-1}{x + 0.01} \right)$$

(4.6.4)

在实际使用中，排斥力趋向于无穷大并不是个大问题。因为在两个物体持续接近的过程中，排斥力逐渐增大，直到超过使两物体相向运动的作用力^①，从而自动地避免了距离趋近 0 的极限条件。

4.6.3 吸引曲线的和

我们可以产生更复杂的曲线，方法是将之前给出的简单曲线累加起来。这样就只需要贮备为数不多的几条简单曲线，而无需收集许多纯粹是自定义或独特的曲线。数量庞大的曲线集合将会是难以处理的，而在已定义的曲线的基础之上构造出来的新的曲线之间则更相关且紧凑。而且，一旦作为基础的低阶曲线之一起了变化，我们能够自动地更新各条复合曲线。

举例来说，我们可以用多个吸引和排斥曲线来表现群体行为。我们可以这样设计简单的群体行为算法：首先为了避免相撞，构造一条在 0 的附近具有高排斥力的曲线；然后在某个最优距离上，排斥力值下降为 0；接下来的一段区域是具有吸引力的；在最远处又逐渐变成互斥。这将标准群体 [Reynolds87] 中的“凝聚度 (cohesion)”和“分离度 (separation)”因素结合成一条曲线。将凝聚度与分离度合并到结果曲线中有一个小的区别，即两者均是基于整个群体的，这要胜过将分离度的测试局限于最接近的个体之间。

① 译者注：直到速度方向改变

当群体中的某些个体之间相距甚远，此时它们之间出现了排斥力。这使得掉队的个体形成自己的小群体。这同时也限制了群体的物理尺寸，因为任何群体若是变得太大了，处在边缘的成员就会被排斥。举例来说，我们可以通过以下步骤得到该行为：将图 4.6.1 与图 4.6.2 相叠加，同时增大铃形曲线的尺度，直到排斥力在远处克服了吸引力，如图 4.6.3。

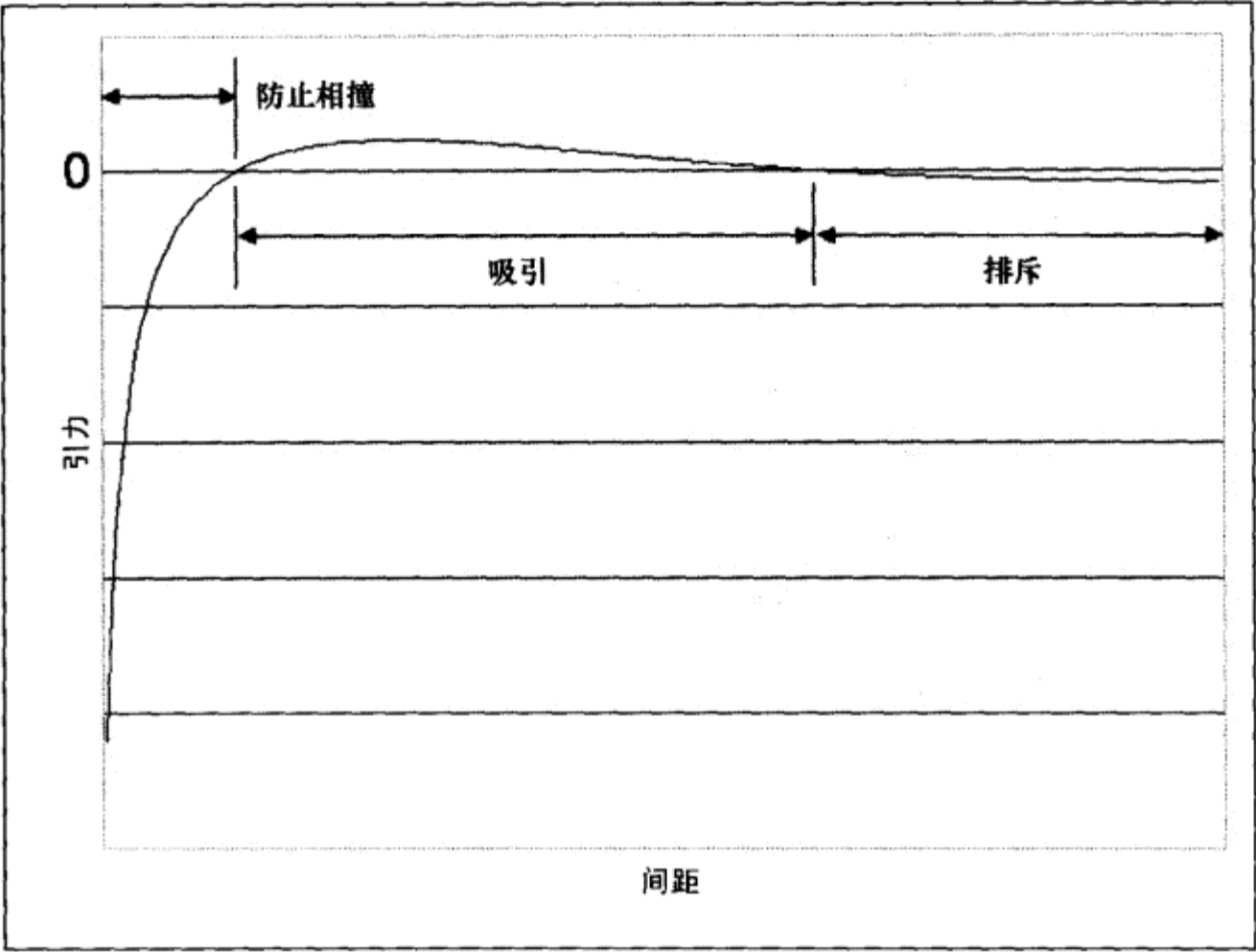


图 4.6.3 $y = -x^{-1} + 4 \times (e^{-x^2})$ 一条简单的群体曲线

4.6.4 对应于特定配对的特定曲线

我们可以在这个简单的基础上构造出更复杂的系统，例如可以令某些特定的游戏对象充当吸引子和排斥子的角色，同时令特定类型的对象只受特定的吸引子和排斥子的影响。

若你的游戏角色中包括狼与兔子，打个比方，兔子应当集结成群，但会从狼的附近逃离。而狼群会集结形成狩猎集团。不过与兔子的恐惧（表现为排斥力）不同，由于饥饿的驱使，狼群会被兔群吸引（表现为吸引力）。

4.6.5 动态曲线

至此，曲线都是恒定的（constant），但事实上没有必要遵守该约束。力是随时间变化的，同时受任意个外部参数的影响。借用狼与兔子的例子，狼的饥饿程度与它不久之前捕食的兔子数量有关，这可以被用来改变狼紧跟在兔群后面意愿的强烈程度。

当一头狼非常饿时，附近的任何兔子都会对其产生强烈的吸引力。而当一头狼吃饱喝足以后，兔子对它产生的吸引力就减弱甚至变成零了。如图 4.6.4，我们可以将整条吸引力曲线

用一个基于饥饿程度的比例系数来缩放，从而产生该行为。图中最上面和最下面的曲线（分别对应比例系数“-0.5x”和“-4x”）均是图 4.6.1 比例缩放后的版本。

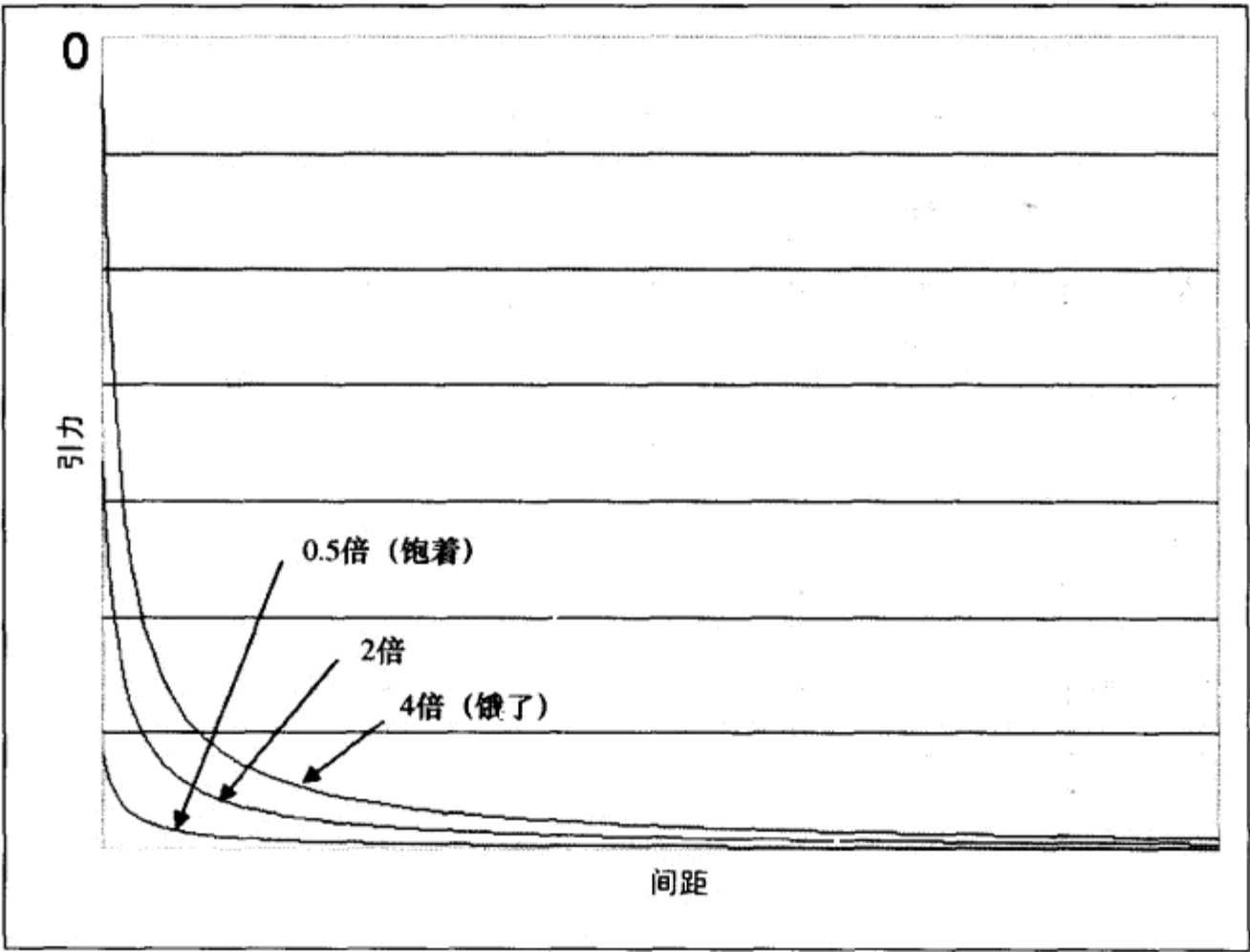


图 4.6.4 兔子对狼的吸引力

我们也可以通过计算多条曲线的加权和来创建动态吸引力曲线，而不是简单地把它们加在一起。公式 4.6.5 表示了从吸引力曲线 f_1 到 f_2 之间的线性插值，其中 w 是范围是从 0~1 的权。相似的，我们也可以对任意数量的吸引力曲线进行加权，只要它们各自的权加起来等于 1。对于现在的两个函数的例子来说， $(1-w)$ 和 w 加在一起正好等于 1。

$$y=(1-w)\times f_1+w\times f_2 \tag{4.6.5}$$

公式 4.6.6 表示了一个较复杂的包含 3 个函数的加权系统。为了正常工作，值 $(w+x+y)$ 必须等于 1 才行。

$$y=w\times f_1+x\times f_2+y\times f_3 \tag{4.6.6}$$

对经过缩放后的图 4.6.1 和图 4.6.2 中表示的函数曲线进行内插，能够产生为数更多的在空间上处于两条原始曲线之间的曲线，如图 4.6.5 所示。图中也画出了均匀地混合两条源曲线所生成的新曲线。如果你希望产生混合曲线位于两条源曲线形成的范围之外，可以进行外插而无需遵守权和为 1 的规则，例如可以使总的权和大于 1。

也可以根据与排斥子或吸引子的相对方向来混合曲线。这需要对系统中的元素增加一个方向属性，不过 AI 的基本数据中基本上总是包括方向属性的。还是举狼和兔子的例子，狼会更大程度地受那些面前能够直接看见的兔子的吸引。可以通过减小位置上处于 AI 角色背后，或位于视锥（cone of vision）之外的目标的比例值来计算出该效果。

同样的变化效果也可以通过动态选用不同的曲线来得到，候选曲线可以基于方向，也可

以基于距离、时刻 (time of day) 或任何其他游戏中的参数。

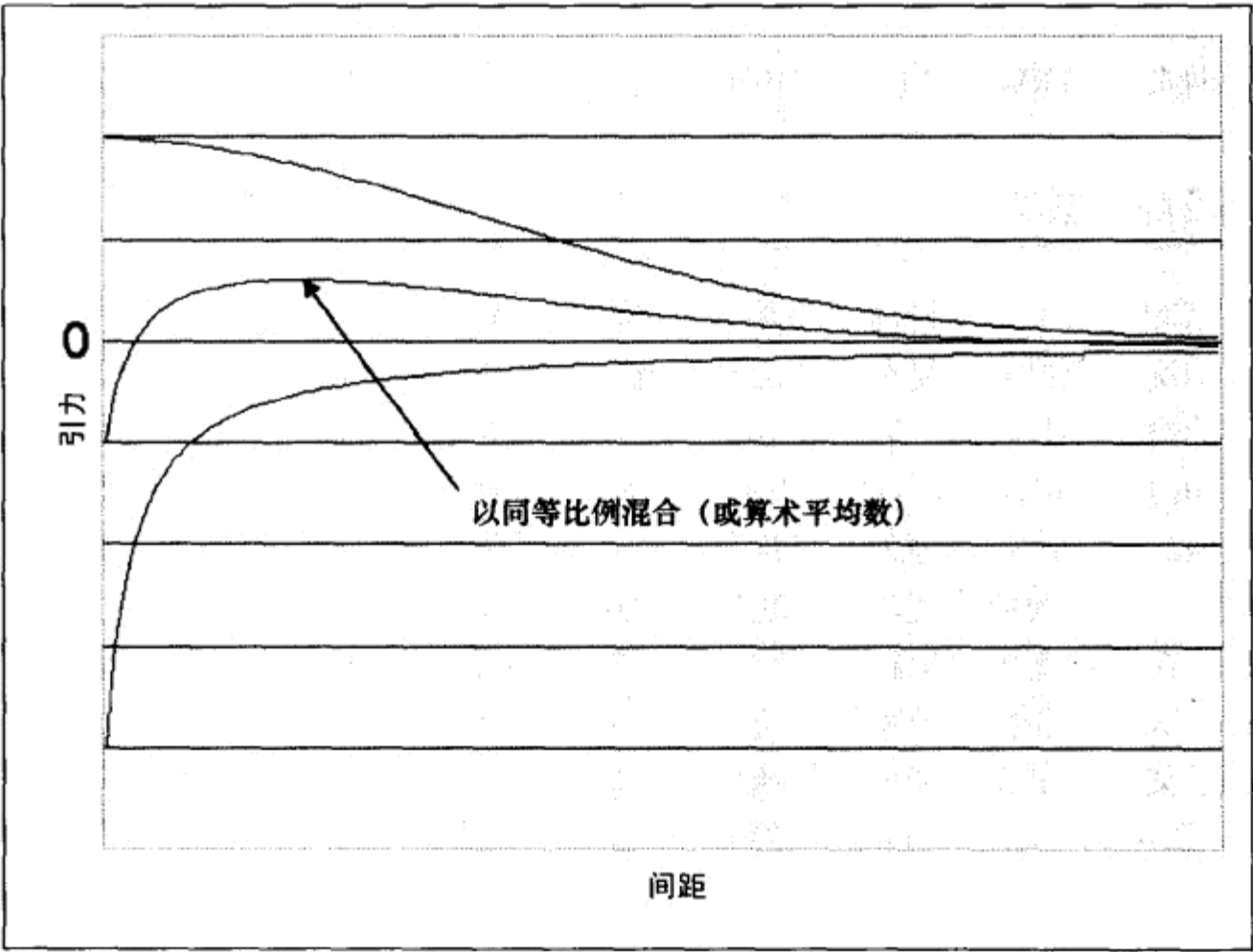


图 4.6.5 内插而得的曲线

4.6.6 点、线、面

到现在为止，我们一直将施力对象用点来表示，但我们不应局限于这一表示法。我们也可以采用更复杂的对象来表示吸引子和排斥子，例如直线、平面、多边形或其他复杂的几何体。不同类型的施力者之间的差异仅仅在于其决定相互作用力所依赖的距离的计算方法不同。简单地计算三维空间中点到点的距离不再适用，我们要确实计算到直线、平面或实体的距离。对于实体而言，这可以是到物体质心 (center of mass) 的距离 (重力)，到物体表面的距离 (静电引力)，或其他根据物体形状及期望行为而定的其他便利的规则。

运用平面可以方便地使对象留在一个约束区域之内。由数个无限大的平面组成的集合可以封闭地形成任意的凸多面体，立方体由于可以很容易地与 xyz 轴对齐，因而是最简单的表示。

由于既可以将平面看作是只有一侧的，也可以看作是具有双侧的，平面表示有一些复杂。对于双面的平面来说，到平面的距离值总是正的，就像点到平面的距离那样。但对于单面的平面来说，到平面的距离值可以是负的。由于在本文的开头我们决定从距离 0 开始画吸引力图像，最简单但同时不破坏该约束条件的方法就是，将所有处于单面平面的反面的物体视为到平面的距离值等于 0。这也是为什么必须保证在距离为 0 的时候，力不能趋向无穷大。要是你忘记了这一点，你的游戏对象就可能突然消失不见，因其获得了无限大的速度。

一个无限大的平面所产生的吸引力矢量的方向总是与该平面的法向量相一致的。直线所

产生的吸引力的方向则总是与该直线的方向垂直并且指向受力的物体的。

像之前提到的那样,为了扩展概念以便使用任意几何体来表示吸引子和排斥子,需要计算到任意几何体的距离。三维对象的相交性测试请参见 *Real-Time Rendering* 一书 [RTR02]。对应此书的网页 [RTRWeb02] 中其中有一节,包含许多交点方法的引用。

4.6.7 AI 控制的层次

当我们试图将吸引子和排斥子与那些将在游戏世界中“聪明地”行走的 AI 角色一同使用的时候,情况会变得非常复杂。在运动系统中有大量需要彼此直接交互的成分。首先是决定该去哪里的高层 AI 代码。它给了我们一个游戏世界中的目标点。然后我们将这信息送至寻路系统(path-finding system),由其决定从当前位置到指定的目的地所应采取的完整的路径。时常需要重做这两个决定,因为在目标移动时,或寻路系统的高层目的变更时,常常会选取新的目的地。一旦目的地改变,或发现现有的路径已被堵死,就需要重建路径。

接下来,在游戏逐帧地运行的同时,移动控制(steering control)开始工作。作为移动控制的一部分,吸引子和排斥子系统负责控制 AI 角色开始沿预设路线运动,并在运动过程中尽可能地接近某条预定义路径。既然速度和方向已知,我们可以将信息传给动画控制系统,使动画与 AI 角色的运动得到同步。最后,轮到物理引擎处理有关的碰撞测试与响应。

让我们考虑赛车游戏的情况。在赛车游戏中,最初只有一条跑道,上面标着起点和终点,分别对应高层 AI 寻路系统的初始位置和目的地位置。接着寻路算法确定从起点到终点间应采取的路径。然后移动控制系统一次一帧地使玩家的车辆沿着路径前进,在避免障碍物的同时试图使车辆按照或至少是靠近之前确定的路径前进。动画控制系统使轮子旋转和偏转,以便赛车看起来像真的一样在赛道上迂回。物理系统保证赛车能随路面起伏并正确地响应其他碰撞。

4.6.8 动画系统的交互

任何对基本运动系统的修改都要与控制 AI 的动画系统相协调。一般来说,飞行或滚动的物体是容易处理的,因为 AI 对象要么并不与地面接触,要么可以找到简单的规则来调整轮子的速度和角度以配合车辆的速度和方向。但在处理循环的走路动画(walking animation cycle)的时候就有不小的困难,必须对动画进行调整,以对吸引子、排斥子或其他移动控制行为对运动系统做出的修改进行补偿。

吸引子和排斥子会修改角色的速度和方向,因此我们要小心地补偿,以便走路的循环有正确的表现。重要的是,要么动态地控制动画的回放速度,或准备多个对应不同速度的动画备用,以此对吸引子和排斥子系统引入的速度变化做出补偿。应当能够简单地控制 AI 对象的运动方向,并仅在每帧中整个移动控制系统得到运行之后才调整朝向。若改变动画回放速度和在多个适应不同速度的动画中切换并不可行,则吸引子和排斥子应当被限制,仅用于修改 AI 对象的朝向。

对寻路进行修改也必须以恰当的顺序来进行,动画和运动才能较好地配合。若我们希望在沿预先计算好的路径移动之前就应用吸引和排斥效果,事后需要稍稍恢复一些由引力产生的效果。若是在计算碰撞之后才进行运动,可能会带来 AI 穿墙的后果。

4.6.9 移动 (Steering)

该技术特别适合作为一项与其他 AI 移动技术配合的改进而使用。一个完全依赖吸引子与排斥子的游戏环境，可能会轻易地因为遇到局部极小值而使物体无法继续运动。因此，最好是采用高层移动及寻路系统来产生高层行为，而仅用吸引子和排斥子来增强这些行为的真实性。

举例来说，在赛车游戏中可以对赛车增加相互作用的排斥力，这样既可以降低赛车之间碰撞的频率，同时又保留了使车辆沿跑道前进的高层控制系统。

同样的机制可以让行人避开接近中的车辆，或使车辆避让静止或移动的障碍物。有选择地设置吸引子和排斥子，在转弯处增加一点吸引或排斥力，能够使赛车原本由直线线段组成的路径变得平滑。但由于无法做出一些高层次的决定，例如遇到岔路时应当选取哪一条分支，还是必须存在一个负责长期策略性决定的适当的高层系统。

在特定的情况下，吸引子和排斥子系统也能在寻路和导航系统的某些方面起辅助作用。举例来说，在角色扮演游戏中，AI 系统可以运用排斥子而使 NPC 避开树或其他简单的障碍物。吸引子和排斥子系统也能够帮助由 AI 控制的角色在从一个房间移动到另一个房间的时候从厅和门的中央通过，同时本能地避开在门口聚集的其他玩家和 NPC。

考虑到游戏本身是一个需要大量 CPU 时间的计算密集型的程序，因此若在系统中使用过多的吸引子和排斥子必会影响性能。你应当有节制地使用它们，或仅考虑处理彼此最接近的对象以便保持最低限度的计算量。吸引子和排斥子也应被置于游戏的空间划分系统中，以便重用所有已有的剔除代码。

4.6.10 结论

可能仅有为数不多的几款游戏能够将本文描述的方法用作其游戏世界中对象移动的主要方法。但是，将本文描述的方法与其他有用的技术结合使用，才能得到最佳的效果。当我们定义了一组吸引力曲线之后，可以将这些简单的效果结合起来，建立一个较为复杂的行为调节器 (behavioral modifier) 的库。

某些系统较容易通过吸引子和排斥子来影响，例如飞行和游泳 AI。同时也有一些系统，因为和那些接触地面的循环走路动画之间有着复杂的互动，所以很难用吸引和排斥的方式描述。但是在恰当的规划之下，这些困难也是可以克服的，可使游戏中的运动更加真实与精细。

4.6.11 参考文献

[Lander99] Lander, Jeff, "Lone Game Developer Battles Physics Simulator," *Game Developer Magazine* (April 1999): pp. 15-18. (译者注:
www.gamasutra.com/features/20000215/lander_01.htm)

[Reynolds87] Reynolds, Craig, "Flocks, Herds, and Schools: A Distributed Behavioral Model," *Computer Graphics: ACM SIGGRAPH Conference Proceedings* (1987): pp. 25-34.

[RTR02] Haines and Akenine-Möller, *Real-Time Rendering, Second Edition*, A. K. Peters Ltd., 2002.

《实时计算机图形学（第2版）》，普建涛译，北京大学出版社2004年7月出版。

[RTRWeb02] Haines and Akenine-Möller, "3D Object Intersection," available online at www.realtimerendering.com/int/, September 26, 2002.



4.7 高级 RTS 游戏造墙算法

作者: Mario Grmani, Sony Online Entertainment
E-mail: mariogrimani@yahoo.com
译者: 肖罡
审校: 李劲松

大多数的即时战略 (RTS) 游戏都用墙或类似的防御性建筑物作为障碍物来阻碍敌军的行进[Pinter01]、[Stout96], 自动的造墙算法能增加非玩家角色 (NPC) 竞争力而且能对随机地图地生成提供有力的支持。一篇关于 RTS 游戏造墙的文章[Grmani03]描述了基本的算法并且讨论了几个潜在的改进, 本文实现了所有那篇文章中建议的改进, 并且涉及了一些更深入的题目, 如, 造城墙, 再利用已有的墙, 可毁坏的自然障碍和建水岸墙。

4.7.1 算法

造墙算法允许我们用智能可控的方式来完成造墙任务, 算法着重于疆域的扩张 (时刻跟踪边界位置), 而不是直接着眼于墙的选址问题。边界位置, 依照算法也就代表了墙的位置, 这种变换是有效的, 因为墙和其保护的领域的确有对应关系。

算法用一个可增长的内部节点链表来实现疆域的扩张, 内部节点表示当前的领土, 扩张时, 一次把一个新内部节点加到链表上。新内部节点是用贪心法[Cormen01]从边界节点中选取的, 这意味着在每一步, 算法检查每一个边界节点, 用启发函数给它们排序, 再从中挑选启发值最小的一个作为新节点并加入链表。贪心法的使用意味者启发函数不得不根据即时、局部的最小值来评估节点。这种方法快速且易于实现, 但是因为启发函数评估方法的“局部”特性决定了不能保证最优解, 但是可以造出近似最优、高质量、符合游戏要求的墙。下边是算法的伪码:

```
List PerimeterList    // 边上的节点
List InteriorList      // 内部的节点
List OutputList        // 这里结果
                        // 退出时, 表中包括需包围的节点

WallBuilder(Node StartNode,
             AcceptanceCriteria Criteria)
{
```

```

Node BestNode, SuccessorNode

clear PerimeterList, InteriorList and OutputList

add StartNode to PerimeterList
while ((PerimeterList is not empty) and
      (Criteria are not met))
{
    use heuristic function to find BestNode
    remove BestNode from PerimeterList
    add BestNode to InteriorList

    for each successor SuccessorNode of BestNode
    {
        if (SuccessorNode is in PerimeterList) or
            (SuccessorNode is in OutputList) or
            (SuccessorNode is in InteriorList) or
            (SuccessorNode is a natural barrier)
            continue

        if (SuccessorNode is at maximum distance)
            add SuccessorNode to OutputList
        else
            add SuccessorNode to PerimeterList
    }
}
move all nodes from PerimeterList to OutputList
}

```

4.7.2 算法改进

现在的算法已经能解决 RTS 游戏中大部分普通的造墙问题，但 RTS 环境的丰富性带来了许多独特的问题，下面的章节就讨论了其中几个有趣的。

1. 海、河和湖

很多 RTS 游戏地图上都有水域，如海、河和湖。墙不能建在水上面，所以水域扮演了和建墙有同样作用的自然障碍的角色，如果游戏不支持基于水上的移动物体，那么水就是一种不可破坏的自然障碍，这个问题已经被上面的算法解决了。

一种更困难的情形是，如果游戏支持可在水上移动的物体，那问题就出来了：水面是可通过的疆域类型（某些物体可以通过它），但不可以在上面建造东西。这是一种新的疆域属性组合，需要我们给出不同的解决方案：沿着水岸线造墙来阻挡敌人越过水域。

最简单的方法来实现建水岸墙就是改变基本算法中处理后续节点的方式。对于后续节点，原来算法直接把它们加到边界列表中，改进算法要检查它们是否邻接水域，邻接水域的节点就直接被加到输出链表中，否则还要被加到边界列表中。这就保证了水岸墙的建造。下边是修改后的伪码：

```
if (SuccessorNode is at maximum distance) or
    (SuccessorNode is adjacent to water) // 增加的新行
    add SuccessorNode to OutputList
else
    add SuccessorNode to PerimeterList
```

不幸的是，上述解决方案有两个主要的缺点，一是非常影响算法的性能，二是它会把所有的水岸都建上墙，尽管有些水岸已经处于更大范围的墙体包围下了。

算法性能上的影响来自于判断是否一个节点邻接水域。例如：在一个8向的格子地图上，当每个边界节点被移到内部节点链表中时，都要产生8个后续节点，增加一个水域邻接检查会增加8个内部循环比较。可以把水岸线作为独特的疆域类型跟踪，就可以避免性能问题，对那些没有这个特性的游戏，我们可以在一开始就找出这些水岸节点，做出标记供后面使用。

对于第二个缺点（疆域内部水域建墙问题），一个解决方法就是增加一个判断：是否这个水岸节点要被加到结果链表中。按照这个方法，当算法运行时，遇到水岸节点就立刻做出判断是否要将墙建在那里，因为此时我们无法知道是否这个水域被疆土完全的包围了，所以无法当场做出正确判断。惟一可行的方案就是用后期处理去掉不必要的水岸墙。

要实现这个方案，我们一开始还要把所有的水岸节点都放到结果链表中；接着根据水域节点，用种子填充法（flood fill）或类似的算法找出地图上所有水域以及水岸节点链表。注意，这样找出的水岸节点可能是也可能不是都在结果链表中，在结果链表中的水岸节点只是所有水岸节点的一个子集。

下一步，一次检验一个水域，看是否它已经被我们的墙体包围。这个水域的所有水岸节点信息是这次测试的关键，如果某个水域所有的水岸节点或者是个自然障碍，或者已在我们的输出链表中，那么这个水域就是已经被墙包围的（内部水域），在结果链表中的所有这个水域的水岸节点都要被标记为候选删除；否则就说明这个水域还没有被墙围起来，相关的水岸节点都要在输出链表中保存下去。

一旦完成了所有水域的测试，我们就可以遍历输出列表，把其中标识为候选删除且不是必须保留的节点都去掉（因为有的水岸节点可能是两个或多个水域共享的，这样会防止误删除）。还有要注意的重要一点，这个算法改进并不只针对于水域障碍，它可以应用到所有类似的疆域类型：不能直接在上边造墙但有允许某些物体通过。

2. 墙的重用

在一个RTS游戏的动态环境中，像墙这样的建筑被部分毁坏是很合理的，同时，随着战略形势的改变，造墙的需求也会跟着改变。在这种情况下，当我们想造新墙的时候，遇到旧墙是不可避免的。要想利用这种情形，我们不得不评估旧墙并将其与待建的新墙相比较：如果旧墙的质量大于或等于新墙，我们就尽量利用它们并它们加到新墙中去。相反，我们就要考虑是否保留旧墙：保留一段质量低的旧墙会给整个墙留下弱点，但是某些时候，比如当我们想用最短的时间造墙时，保留旧墙也是有益的。要实现重用旧墙，我们需要对原来的算法做几处修改。

首先，我们要修改启发函数使其能利用旧墙：当一个边界节点与旧墙邻接时，这个节点的启发值要被调整。通常这意味着降低这个节点启发代价（受益于旧墙），但有些时候，代价反而会提高，比如旧墙质量不可接受。表 4.7 的例子给出了不同的旧墙质量及相应的启发代价。

表 4.7.1 一个根据不同障碍类型分配启发代价值的例子

阻碍类型	启发代价值	资源代价
不可破坏的自然资源	0	0
已存在的更高质量的墙段	50	0
已存在的同等质量的墙段	75	0
新的墙段	100	50
已存在的更低质量的墙段	125	0
已存在的不可接受质量的墙段	不可知或 100	不可知、0 或反款

不可毁坏的自然障碍是最理想的障碍物，所以我们把它们启发代价设为 0，建新墙的启发代价作为其他方式的参照物，方便起见定为100。与新墙有相当质量的旧墙，其启发代价要小于新墙，这样能确保算法倾向于选择利用旧墙来建新墙；比新墙质量更好的旧墙，其启发代价就更低，能确保其有仅次于天然障碍物的优先权；最后，为了不鼓励使用比新墙质量要低的旧墙，我们将它们启发代价设得比建新墙还要高。

低质量的旧墙需要特别的处理，我们或者绕过它们建新墙，或者拆除它们再建新墙。第一个选择类似前面对水域的处理，但不是很高效。此时，低质量旧墙节点不会被考虑成边界节点，也就不需要计算启发代价值了，这也就是为什么在表 4.7.1 中（最后一行），它的启发代价为“未知”。

第二种选择，在旧墙的地址上直接建造一个新墙，会更好一些，但需要更多的工作。删除旧墙需要一个新的链表——删除链表，里面存放这要被新墙所替代的旧墙节点。当算法遇到低质量的旧墙时，会当它们不存在而把它们加到新墙候选地址链表，与此同时，旧墙节点都要被加到删除链表。这样节点的启发代价和建新墙的启发代价差不多：如果移除旧墙会很快而且得到一些反款，我们也许可以降低它们启发代价；相反，如果移除旧墙很慢，我们可能会增加其启发代价。

还要记住重要的一点，尽管表 4.7.1 中启发代价的相对值是准确的，它们的绝对值要视情况而定。任何一个启发式方法，我们都需要作些试验来设定出这些特定的值以便得到理想的结果。启发代价表可以用一个查找表来实现，表中是各种障碍物和其对应的启发代价值。因为这样的一张表很大程度上控制了算法的行为，让调用造墙算法的程序设置这些值是很有帮助的，这样，算法的调用程序可以定制每一个造墙请求。

3. 可被破坏的自然障碍物

到现在为止，我们一直假设自然障碍物是不可破坏的，因而它们的启发代价也是最低的，但是有些 RTS 游戏有可以被破坏的自然障碍物，所以上述假设并不适用所有情况。我们能想象到，可被破坏的自然障碍物当然没有不可被破坏的好，这应该反映在它们的启发代价值上。表 4.7.2 给出了一个例子：不同类型自然障碍物及其相对应的启发代价值。

表 4.7.2 一个根据不同自然障碍类型分配启发代价值的例子

自然屏障类型	启发代价值	资源代价
不可破坏的自然障碍	0	0
坚固的可破坏的自然障碍	20	0
中等强度的可破坏的自然障碍	40	0
很弱的可破坏的自然障碍	60	0
不可接受质量的可破坏的自然障碍	不可知或 100	不可知、0 或奖励

不可被破坏的自然障碍物依旧得到最低的启发代价值，其他可被破坏的其启发代价值与其坚固程度成反比。因为表 4.7.2 和表 4.7.1 都是用在同一个启发计算函数中，所以它们有着相同的刻度。为了示范交叉比较，表 4.7.2 中很容易被破坏的自然障碍物的启发代价值被设成高于表 4.7.1 中更好质量旧墙的代价值，也说明在有些情况下，旧墙比可被破坏的自然障碍物更好。

对具有不可接受质量的可被破坏自然障碍物的处理，可类似于前面对不可接受质量的旧墙处理，表 4.7.2 中最后一行的启发代价值和资源消耗值就反映了这种相似性。移除有不可接受质量的自然障碍物从而得到一些奖励（可以是采集到一些资源），这同上边的拆除旧墙得到反馈是一样的。具体实现这个启发计算过程时，表 4.7.1 和表 4.7.2 中所有的启发代价值都被存在一张查找表中，这张表在每个造墙请求前可以被定制。

4. 造城墙

RTS 游戏通常要有很多的建筑物和其他静态的东西聚集在一起，组成村庄和城镇，这些聚集体需要被保护起来，扩展上面的造墙算法来包围这些聚集体是很有用的。

我们可以用非常简单的穷举法来实现城墙的建造。先找到一个大致的聚集体中心点作为起点，然后计算出中心点到聚集体中最远的物体的距离，这个距离就被传到造墙算法中作为最短距离造墙，能确保所有的物体都被包围在墙内。这种方法可以工作，但是非常浪费。它不正确地假设城市里的物体是按起始点对称分布的，在某些方向上，这个最小距离可能会让墙离真正的物体非常远，结果就是造墙代价比实际需要昂贵得多，城市内部面积也比需要的大得多。

有一个更好的但也复杂很多的方法，它先生成能包含所有需要的物体的最小地形，这个地形然后作为起始点被输入到造墙算法中。下面是建造初始墙的伪码：

```
List PerimeterList    // 边上的节点
List InteriorList     // 内部的节点
List MinInteriorList  // 最小内区域所包含的节点
List OutputList       // 部分结果

CreateInitialWall(Node StartNode, List ObjectList)
{
    Node ObstructedNode, NextNode, SuccessorNode
    Object CurrentObject
    reset PerimeterList, InteriorList,
        MinInteriorList and OutputList
```

```

// 将对象的轨迹增加到最小内区域中
for each object CurrentObject in the ObjectList
    for each node ObstructedNode obstructed by
        CurrentObject
        add ObstructedNode to MinInteriorList

// 将轨迹中间的路径增加到最小内区域中
for each object CurrentObject in the ObjectList
{
    find path between StartNode and CurrentObject
    add all path nodes to MinInteriorList
}

// 创建第一堵墙
while (MinInteriorList is not empty)
{
    remove node NextNode from MinInteriorList
    add NextNode to InteriorList
    for each successor SuccessorNode of NextNode
    {
        if (SuccessorNode is in PerimeterList) or
            (SuccessorNode is in OutputList) or
            (SuccessorNode is in InteriorList) or
            (SuccessorNode is in MinInteriorList) or
            (SuccessorNode is a natural barrier)
            continue

        if (SuccessorNode is at maximum distance) or
            (SuccessorNode is adjacent to water)
            add SuccessorNode to OutputList
        else
            add SuccessorNode to PerimeterList
    }
}
}

```

退出时，perimeter list、interior list 和 output list 保存着必须要被传到造墙算法主程序循环的数据，主循环也要做相应的修改以防止在初始化过程中将这 3 个链表清空。因为主程序要从这些初始化的数据开始工作，原来算法中的起始节点就不必要了，即使在初始化的代码中，起始节点也要被上面选中的对象中的一个代替作为起点。如果需要一个更优的连接集合，我们可以用 Kruskal 或 Prim 的算法来生成一棵最小树。

4.7.3 输出链表的形式

在游戏中，当 AI 玩家需要用输出链表来造墙时，把输出链表转换成易于使用的形式是很必要的。具体的形式取决于发出造墙命令的 AI 玩家，通常这意味着按邻接关系将节点排序并把链表分割成若干小且易于管理的节点组。

游戏中，每一个排好序的节点组都是非玩家角色（NPC）发出的造墙指令的一部分，对使用建筑工人的 RTS 游戏来说，命令的形式和优化的步骤都是非常重要的：建筑工人接到命令，要走到实际的造墙地址，所以让他们尽可能高效率地移动和工作是非常重要的。

4.7.4 结论

在这篇文章里，我们探索了一个造墙算法的一些高级特性和新的想法。它们中的一些，比如处理水域和可被破坏自然障碍物问题，涉及到了更复杂的疆土特性；还有一些，像旧墙的再利用和造城墙，改善了原来算法的效率并且扩展了功能。这篇文章可以作为更先进的 RTS 造墙算法的起点。

4.7.5 参考文献

[Cormen01] Cormen, Thomas H., et al., *Introduction to Algorithms, Second Edition*, MIT Press, 2001.

[Grimani03] Grimani, Mario, "Wall Building for RTS Games," *AI Game Programming Wisdom 2*, Charles River Media, 2003.

[Matthews02] Matthews, James, "Basic A* Pathfinding Made Simple," *AI Game Programming Wisdom*, Charles River Media, 2002.

[Pinter01] Pinter, Marco, "Toward More Realistic Pathfinding," *Gamasutra*, available online at www.gamasutra.com/features/20010314/pinter_01.htm, March 14, 2001.

[Stout96] Stout, Bryan, "Smart Moves: Intelligent Pathfinding," *Game Developer Magazine*, October 1996, available online at www.gamasutra.com/features/19970801/pathfinding.htm.



4.8 利用可编程图形硬件处理人工神经网络

作者: Thomas Rolfes

E-mail: tr@circensis.com

译者: 沙鹰

审校: 李劲松

人工神经网络 (Artificial Neural Network、ANN) 模仿生物学的信息处理, 广泛运用于需要从输入非线性映射到输出集的应用程序中。在实时系统中进行测定 (evaluation) 及有关的网络训练 (network training) 通常对计算的要求很高。Chellapilla 和 Fogel 将一个小的人工神经网络演化达到了专业跳棋选手的水准 [Chellapilla00]。这在一台 400MHz 的 Pentium II 个人电脑上用了超过 6 个月的时间来运行 840 代的演化, 不过程序没有特别进行优化。

第二代可编程图形处理器 (Graphics Processing Unit, GPU) 引入了单精度和半精度的浮点纹理模式、浮点像素流水线 (pixel pipeline)。直到不久以前, 在消费级图形硬件上进行神经网络模拟还只限于利用固定的 8bit 混合, 通过 CPU 实现则要快得多。今日, GPU 的向量处理性能已经远远超出标准 PC 和视频游戏机的 CPU, 其指令及存储器的吞吐量均高出一个数量级。在大批应用程序中, GPU 正迅速发展成为完全可编程的向量协处理器 [GPGPU]。

人工神经网络的高效实现有赖于用于科学计算的数值线性代数库, 如 BLAS [Lawson 79]。关于将 BLAS 移植到 GPU 已经取得了初步成果, 深入的工作正在进行中。将 GPU 作为通用向量处理器来使用, 程序员们可以把原先由 CPU 执行的向量化例程交由 GPU 执行, 从 GPU 的高性能中获益, 并更好地均衡负载。

本文展示了如何通过以下两者来实现一个人工神经网络: 基于 GPU 的 BLAS3 风格的单精度矩阵—矩阵积 (SGEMM) [Dongarra88], 和以 Direct3D 9 实现的激励函数 pixel shader。^{*}

4.8.1 CPU 与 GPU 系统架构

Flynn 分类法 [Flynn72] 是按计算机内的指令流和数据流是单个还是多个来划分计算机种类的方法。现有的游戏机和个人电脑的 CPU 中一部分

^{*}注: 在 OpenGL 术语中称作 “Fragment shader”。

具有超标量 SISD 内核（单指令流、单数据流），有些还具有浮点 SIMD（单指令流、多数据流）的向量扩展，例如 PC 和 Xbox 的 CPU 所采用的 Intel SSE，又如 PowerPC 采用的 AltiVec 技术（不包括 GameCube 使用的 PowerPC 750 CPU），以及 PlayStation2 中的向量协处理单元（VU）。平台则或采用统一内存架构（Unified Memory Architecture, UMA），如 Xbox；或采取非统一的 NUMA 与高速 DMA 通道合作的方式以便连接各子系统，如 GameCube、PS2 还有 PC。

GPU 中的可编程组件就是 SIMD 的 vertex shader 和 pixel shader 单元。vertex shader 单元读入顶点流，并执行 vertex shader 程序。然后颜色值、纹理坐标和其他数据进行插值并传到 pixel shader 单元中，由后者执行 pixel shader 程序来查询纹理元素以及按其他输入数据进行计算。shader 程序可以是汇编风格的，也可以用高级语言写就。当前的高端 GPU 具有 4 个 vertex shader 单元、8 个 pixel shader 单元、全并行处理。shader 单元通过高速缓存和宽的总线访问显示内存。

PlayStation2 的向量单元可以通过编程起到人工神经网络中的向量/矩阵内核的作用 [PS2Neural]。也可以使用 vertex 程序进行通用流处理，而 Xbox 的 GPU 上的 vertex 状态 shader 允许你不断地修改常数寄存器。未来的游戏主机一定更适合用来做高速通用的向量计算。

4.8.2 人工神经网络

外反馈网络（feed-forward network），又称作多层感知器（multilayer perceptron），可能是人工神经网络用于受控学习（supervised learning）最常见的结构。建议初次接触神经网络的读者阅读《游戏编程精粹 1》中的入门 [LaMothe00] 和《游戏编程精粹 2》中的例子 [Manslow01]。

为了计算 N 层外反馈网络的线性基函数，需要计算：

$$a_{i+1} = \text{act}(a_i \cdot W_i) \quad (4.8.1)$$

其中 act 是一个激励函数； W_i 是加权矩阵；向量 a_1 是输入层，包含输入数据节点和一个可选的偏差（节点； $a_2 \dots a_{n-1}$ 是内部隐藏的层； a_n 则是输出层。通常，层与层之间向量的尺寸和加权矩阵不相同。

在 m 个输入集的并行计算情况下，向量—矩阵积就变成了矩阵—矩阵乘法

$$A_{i+1} = \text{act}(A_i \cdot W_i) \quad (4.8.2)$$

这 $n-1$ 个矩阵 A_i 的行数为 m 。广义激励函数接受矩阵作为参数。为了实现非线性映射，使用非线性激励函数，如下：

高斯函数： $\exp(-a^2 \sigma^{-2})$

Hardy 双二次函数： $\text{sqrt}(a^2 + \sigma^2)$

双曲正切函数： $\tanh(a) = (e^a - e^{-a}) / (e^a + e^{-a}) = \tanh_2(a / \ln(2))$

Sigmoid 函数： $(1 + \exp(-a/\sigma))^{-1}$

有一些高效的矩阵—矩阵内核，如 ATLAS 中优化过后的 BLAS3 例程 [Whaley98]，通过分段（partitioning into submatrices）来提高缓存的重用率。该方法称为矩阵分块相乘（block matrix multiplication）。

4.8.3 实现

有一些作者, 包括 Larsen & McAllister [Larsen01]、Moravanszky [Moravanszky03] 和 Krüger & Westermann [Krüger03], 撰文探讨了如何高效地实现基于 GPU 的稠密矩阵及带状矩阵乘法。基本概念是将矩阵表示为纹理, 选取合适的顶点坐标 (vertex coordinate) 和纹理贴图坐标 (texture coordinate) 后通过 vertex shader 将其渲染为四边形。插值后的贴图坐标传入 pixel shader 单元, 由其在渲染目标上实际执行并行的乘积累加 (multiply-accumulate) 操作。有的实现需要 pixel shader 后期混合 (post blending), 而在某些仅支持浮点缓冲的硬件上无法实现此项功能。混合在 pixel shader 中进行, 需要两张纹理作为输入, 另外还需要第三张纹理作为渲染目标。这一过程可以通过滚动源和目标来重复执行。稀疏矩阵的乘法可以通过查找表来进行, GPU 实现亦会因图形系统的高内存带宽和低访问迟延而获益。

矩阵乘法计算完成后, 作为结果的渲染目标就变为激励函数的输入纹理。再一次, 在整个矩阵表面上渲染出一个四边形, 或当矩阵大小超过最大允许的纹理大小时渲染出多个四边形。对神经网络上的每一层都重复一次矩阵乘法和后继的激励。下面就给出直接了当地实现双曲正切激励函数的 pixel shader 汇编程序:

```
ps_2_0                // shader 版本
dcl_2d s0              // 纹理阶段
dcl t0.xy              // 纹理坐标

texld r0, t0, s0       // r0 <- texel.xyzw
mad r0, r0, c0.x, c0.y // r0 <- r0*scale+bias

// 双曲正切 (以 2 为基)
exp r1.x, r0.x         // r1 <- 2^r0
exp r1.y, r0.y
exp r1.z, r0.z
exp r1.w, r0.w
exp r2.x, -r0.x        // r2 <- 2^(-r0)
exp r2.y, -r0.y
exp r2.z, -r0.z
exp r2.w, -r0.w
add r3, r1, r2         // r3 <- r1+r2
sub r4, r1, r2         // r4 <- r1-r2
rcp r3.x, r3.x         // r3 <- 1/r3
rcp r3.y, r3.y
rcp r3.z, r3.z
rcp r3.w, r3.w
mul r0, r3, r4         // r0 <- r3*r4

// 将结果写入输出寄存器
mov oc0, r0
```

虽然双曲正切函数的值域是在 $-1 \sim 1$ 之间, 使用 pixel shader 2.0 语言里的 min 和 max 指令时可以更清楚地限制范围。

```
max r0, r0, c0.z      // r0 <- maximum(r0, lower)
min r0, r0, c0.w      // r0 <- minimum(r0, upper)
```

可以通过用指令修饰后缀“_sat”将结果限制在0~1之间。“_sat”后缀对除frc、sincos以外的算术指令均有效，并且使用后缀并不增加指令长度。

```
mul_sat r0, r3, r4
```

本文给出的代码是用GPU实现的Chellapilla和Fogel两人的跳棋位置——处理一个外反馈网络。在ATI Radeon 9700 Pro上的运算明显快于Pentium4 3GHz上运行的特别为SSE优化过的SGEMM版本。对于大矩阵来说，像Strassen [Strassen69] 和Winograd [Winograd68] 那样递归地执行算法，比起简单地进行子矩阵相乘来说，效率又提高了不少。

4.8.4 结论

用GPU实现比起光用CPU来，效率提高不少。使用目前的图形硬件，可以处理容量约为 10^6 个节点， 10^8 个权，平均连通度为100的网络；还是远远低于人脑里的神经元数量 10^{11} 、神经键数量 10^{15} 。随着模拟硬件日益强大，网络的组织结构和学习算法才是人工神经网络研究领域的真正挑战。

4.8.5 参考文献

在线版论文的链接更新：www.circensis.com/gg4.html。

[Chellapilla00] Chellapilla, K., and D. B. Fogel, “Anaconda Defeats Hoyle 6-0: A Case Study Competing an Evolved Checkers Program against Commercially Available Software,” *Proceedings of the 2000 Congress on Evolutionary Computation*, IEEE Press, Piscataway, NJ, pp. 857–863, available online at www.natural-selection.com/NSIPublicationsOnline.htm.

[Dongarra88] Dongarra, J.J., J. Du Croz, S. Hammarling, and R. J. Hanson, “An Extended Set of FORTRAN Basic Linear Algebra Subprograms,” *ACM Trans. Math. Soft.*, Vol. 14 (1988), pp. 1–17.

[Flynn72] Flynn, M., “Some Computer Organizations and Their Effectiveness,” *IEEE Trans. Computers* Vol. 21, 9 (1972), pp. 984–960.

[GPGPU] “General Purpose Computing Using Graphics Hardware,” available online at www.gpgpu.org.

[Krüger03] Krüger, J. and R. Westermann, “Linear Algebra Operators for GPU Implementation of Numerical Algorithms,” *SIGGRAPH 2003 conference proceedings*, available online at www.cg.in.tum.de/Research/Publications/LinAlg.

[LaMothe00] LaMothe, A., “A Neural-Net Primer,” *Game Programming Gems*, Charles River Media, 2000.

[Larsen01] Larsen, E.S. and D. McAllister, “Fast Matrix Multiplies Using Graphics

Hardware,” *Super Computing 2001 Conference*, Denver, CO, November 2001. Available online at www.sc2001.org/papers/pap.pap313.pdf.

[Lawson79] Lawson, C., R. Hanson, D. Kincaid, and F. Krogh, “Basic Linear Algebra Subprograms for FORTRAN Usage,” *ACM Trans. Math. Software* Vol. 5 (1979), pp. 308–371.

[Manslow01] Manslow, J., “Using a Neural Network in a Game: A Concrete Example,” *Game Programming Gems 2*, Charles River Media, 2001.

[Moravanszky03] Moravanszky, Á., “Dense Matrix Algebra on the GPU,” to appear in *ShaderX²*, Wordware Publishing, 2003, available online at www.shaderx2.com/shaderx.pdf.

[PS2Neural] “PS2 Neural Network Simulator,” project site online at <https://playstation2-linux.com/projects/ps2neural>.

[Strassen69] Strassen, V., “Gaussian Elimination Is Not Optimal,” *Numerische Mathematik*, Vol. 13 (1969), pp. 353–356.

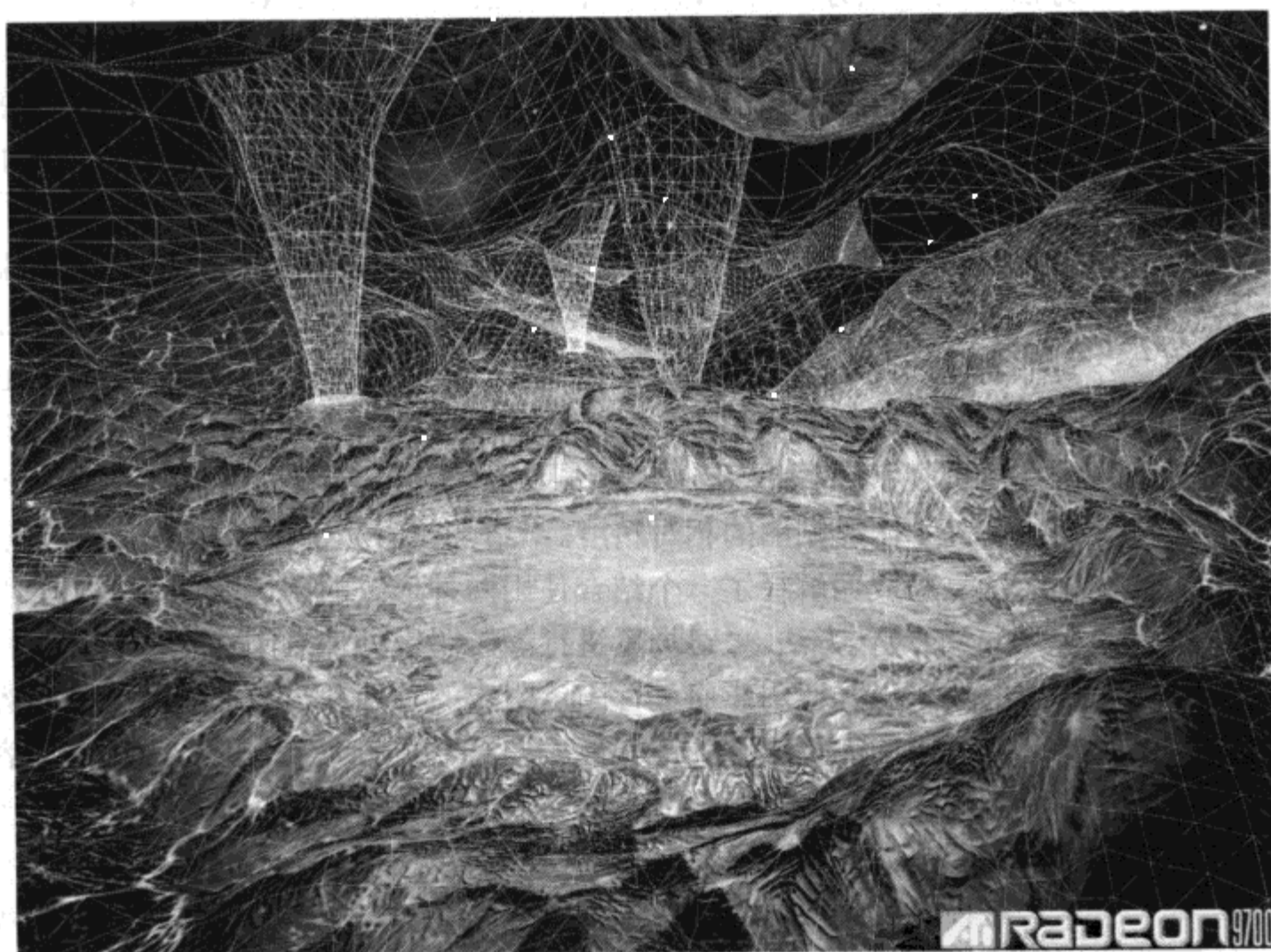
[Whaley98] Whaley, R.C., and J. Dongarra, “Automatically Tuned Linear Algebra Software,” *Super Computing 1998 Conference*, Orlando, FL, November 1998.

[Winograd68] Winograd, S. “A New Algorithm for Inner Product,” *IEEE Trans. Computers*, C-17:693–694, 1968.



第 5 章

图形图像



新学网
PDG

简 介

作者: Alex Vlachos, ATI Research, Inc.

E-mail: Alex@Vlachos.com

译者: 刘永静

审校: 沙鹰

就像图形硬件设备的性能日益增强一样, 游戏中图形图像的品质也将日益提高。图形硬件的更新非常迅速, 即使是今日最新的图形硬件, 过一阵子也同样会让开发者对其失去兴趣。考虑到本书应当在未来几年里都对大家有所帮助, 我们为图形部分挑选的这 15 篇文章并不局限在现有的图形硬件上。尽管文中的例子源代码和 Shader 代码都是以现有的 API 写的, 但你还是可以发现, 这些文章的核心部分是一些有用的方法, 而这些方法的生命周期将远比图形硬件或 API 要长远。

本部分的第一篇文章是“海报质量的屏幕截图”, 这是篇对曾在《游戏编程精粹 2》上介绍过的相应方法加以改进的文章。截取高分辨率的屏幕截图是开发流水线的一个重要组成部分, 本文为此提供了一个极好的方法。

阴影 (Shadow) 一直以来都是图形界的热门话题。当今图形硬件的功能已经够强, 给予了开发者对许多已经公诸论文的阴影算法进行实验的自由。本部分约有三分之一的文章是跟阴影有关的, 总共有 5 篇。“用 GPU 对非封闭的网格模型生成容积阴影”一文提供了一个很好的方法, 它用于对那些存在裂缝的模型生成容积阴影 (Shadow Volume)。“透视阴影贴图”则在后透视空间 (Post-perspective Space) 里执行大家所熟悉的阴影贴图算法, 提高了阴影贴图的相对精度。“结合使用基于深度和基于 ID 方法的阴影缓冲”为我们解释了一个结合使用两种形式的阴影贴图 (基于深度和基于 ID) 的最佳方法。“对场景进行静态阴影处理”阐述了如何通过直接在已有的场景里开辟阴影而进行固定阴影的预处理, 从而避免了使用容积阴影计算带来的大量填充开销。最后一篇以阴影为主题的文章是“为容积阴影和预优化的网格模型调节实时光照”, 此文探索了在多数阴影算法所需要解决的一些人为光照方案。

对最终渲染所得的图像进行后期处理 (Post-Processing), 在游戏中正逐渐成为通行的做法。我们这里也有几篇文章专门讨论后期处理及其他影响整体画面的技术。它们可应用于全部的图像。“实时半色调化 (Real-Time Halftoning)”展示了一个非常有趣的非照片真实级 (Nonphotorealistic) 的风格。“快速 Sepia 色调转换”通过使用 YUV 色彩空间, 将通常的 RGB 图像转换为 sepia 色调, 使图像看上去犹如发黄褪色的老照片一般。“基于场

景照明取样的动态 Gamma”展示了一个有用的方法,可以模仿瞳孔在感受到环境光线强度变化时自动调节的功能。“热空气和薄雾的后期处理特效”探讨了一种模拟由热导致的视觉模式的方法。

本部分余下的文章涉及了各种各样的主题。“将三维模型以所属团队的代表色着色的方法”一文比较了在多人游戏里为团队成员着代表色的数种方法。“使用 Quaternion 的硬件 Skinning”提供了一个使用 vertex shader 对带有皮肤的对象进行处理的好方法。“动作捕捉数据的压缩”对于那些要与量大得简直无法处理的动作捕捉数据打交道的人们来说是非常有价值的。“快速碰撞检测”展示了一个检测骨骼层次碰撞的分级别的方法。“用地平线的地形遮挡剔除”介绍了一个剔除掩盖在山丘或者建筑物后面的地形的非常好的方法。

能负责本部分文章的编辑是我的荣幸。所有文章的核心方法都是很有意义的,我衷心希望读者和我一样,从中获益匪浅。



5.1 具有海报质量的屏幕截图

作者: Steve Rabin, Nintendo of America Inc.

E-mail: steve@aiwisdom.com

译者: 刘永静

审校: 谷超

在游戏的推广过程中,我们常常会需要使用屏幕截图来制作一些广告,箱子的包装、预告文档、攻略指引、杂志封面、大尺寸的宣传海报等。不幸的是,由于原始的屏幕截图质量不高,如果不经处理,就很难达到这些需求所规定的要求。例如,在媒体印刷业,典型的版面布局需要300dpi,这就是说,640×480的控制台屏幕截图在报纸上显示,只有 $5.08 \times 3.81 \text{cm}^2$ 大小。除此之外,一些在电视上看起来效果很好(因为电视有点模糊)的视频游戏,当将相同的原始画面搬到计算机显示器上显示的时候,却产生了很明显的锯齿。由于这些原因,我们非常需要改善屏幕截图的质量,以便应用于各种用途。

为了渲染出具有海报质量的屏幕截图,我们必须从以下两个方面入手:

- 提高分辨率;
- 提高像素质量(抗锯齿)。

5.1.1 提高分辨率

有一种非常简单的方法可以提高超过帧缓冲尺寸的屏幕截图的分辨率。实际上,这个问题就如同使用傻瓜式照相机去捕捉大峡谷的全景图。该技术将多张图并排排列,然后将它们合成到一起形成一张单一的大图。对游戏屏幕截图来说,可以考虑使用这种方法将一些贴图合成到一起形成一张单一的大图,如图5.1.1所示。要做到这一步,我们必须小心地转换这些子图某些地方的平截头体(frustum),让它们成为带有惟一的射影矩阵的贴图,以使这些图像看起来好像是在边缘处刚好吻合。该技术的详细论述参见[Vlachos01]。

除了图5.1.1中的方法以外,我们还可以通过另一种方法来提高分辨率,它可以为我们提供一些图5.1.1中的方法所不能提供的优点。这项改良的技术虽然也需要截取若干贴图,但是每张贴图只需要通过像素的一小部分就可以切换它的视口(使用标准的SetViewport指令而不是转换射影矩阵)。一旦所有已被切换过的贴图都被截取以后,它们就可以合成到一起,形成一张单一的高分辨率的大图,就像图5.1.2所示的插图。

这项只对局部像素样本切换视口的技术主要有三个好处。

- 跟图形管道相比，相对比较简单，并且与其无关。
- 将切换技术所引起的裁剪问题减到最小。
- 考虑到了对任意点都要抗锯齿（在下一部分中讨论）。

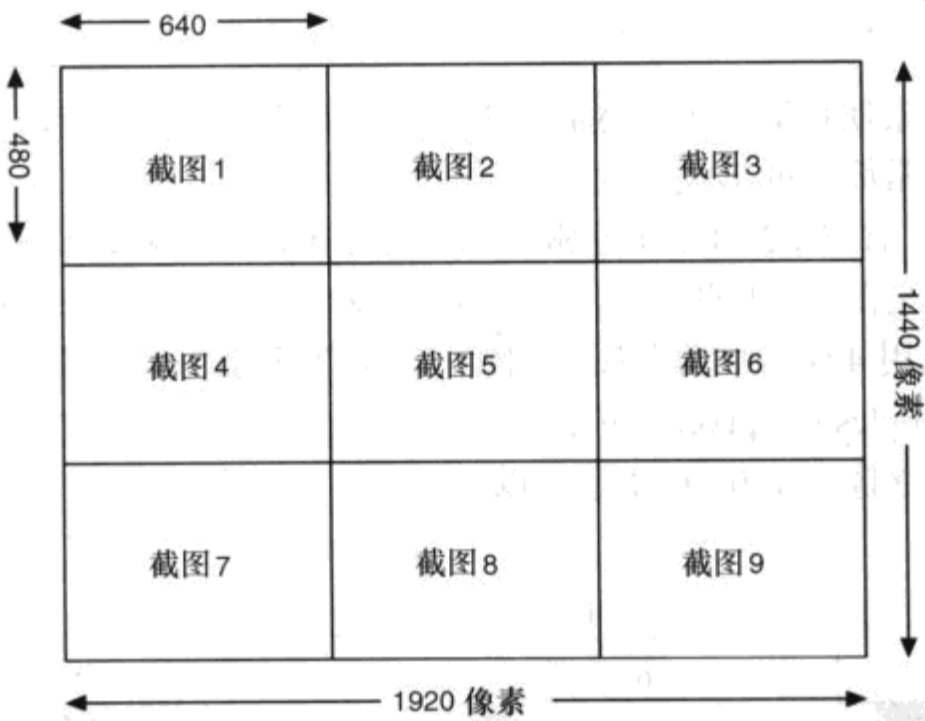


图 5.1.1 合成单个屏幕截图来提高分辨率的转换方法。在这个例子中，截取 39 个截图，将分辨率从 640×480 提高到 1920×1440，从而图像被放大 39 倍

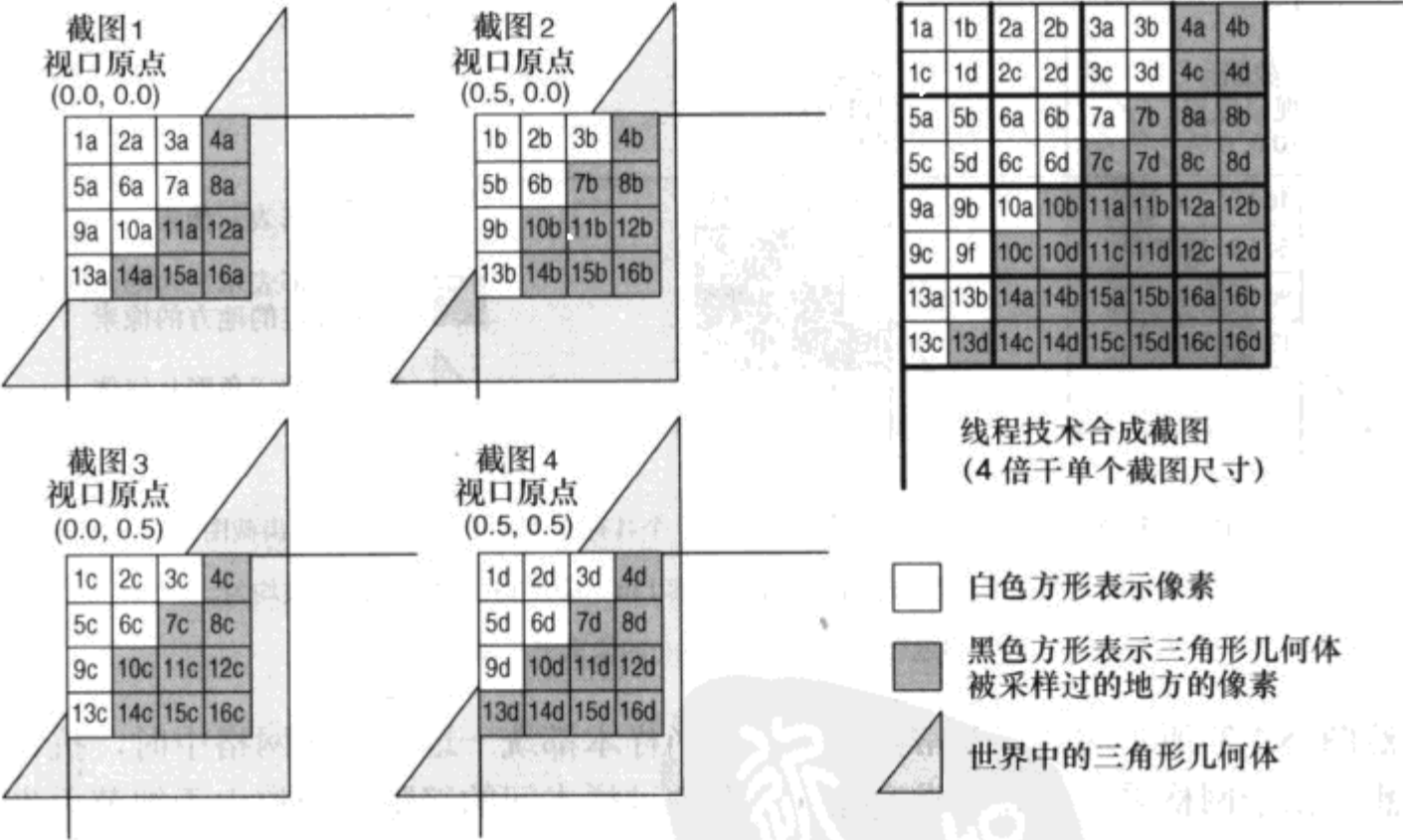


图 5.1.2 截取了 4 个屏幕截图，每张截图都有自己的经过像素的一小部分转换过的视口。三角形代表了那些被采样过的和栅格化过的世界中的几何体（样本采自每个四边形像素的左上角）。右边是最终的合成截图，它的分辨率是原先单个截图的 4 倍。注意每一个源截图像素是如何被放置在合成截图中的（例如，源截图中的像素 13a 被放置在合成截图中的 13a）。

这项视口切换技术中所引起的一个问题是，必须调整 mipmap 斜线以说明详细的采样。依靠贴图，你或许会想要倾斜 mipmap 的水平面，以便总是可以使用高分别率的纹理，或者简单地使 mipmapping 完全不可用。

5.1.2 提升像素质量

抗锯齿是提升像素质量的关键。然而，很多游戏并不看重抗锯齿，或者虽然它们这样做了，但是却使用了非常简单的形式。视口切换技术之所以令人兴奋，是因为它可以用来获得非常高质量的抗锯齿。图 5.1.3 显示了非常多的屏幕截图，每张贴图都通过其某个像素的一小部分被切换（使用的是图 5.1.2 的方法），它们可以被组合成一张单一的抗锯齿的图像。在图 5.1.3 的情况下，可以用 4 张源屏幕截图来创建一张同等大小的屏幕截图，此屏幕截图在逐像素 4 个样本（four samples per pixel）下是抗锯齿的。这项技术可以被推广应用于任何数量级的抗锯齿，即使超过逐像素 1000 样本也可以。

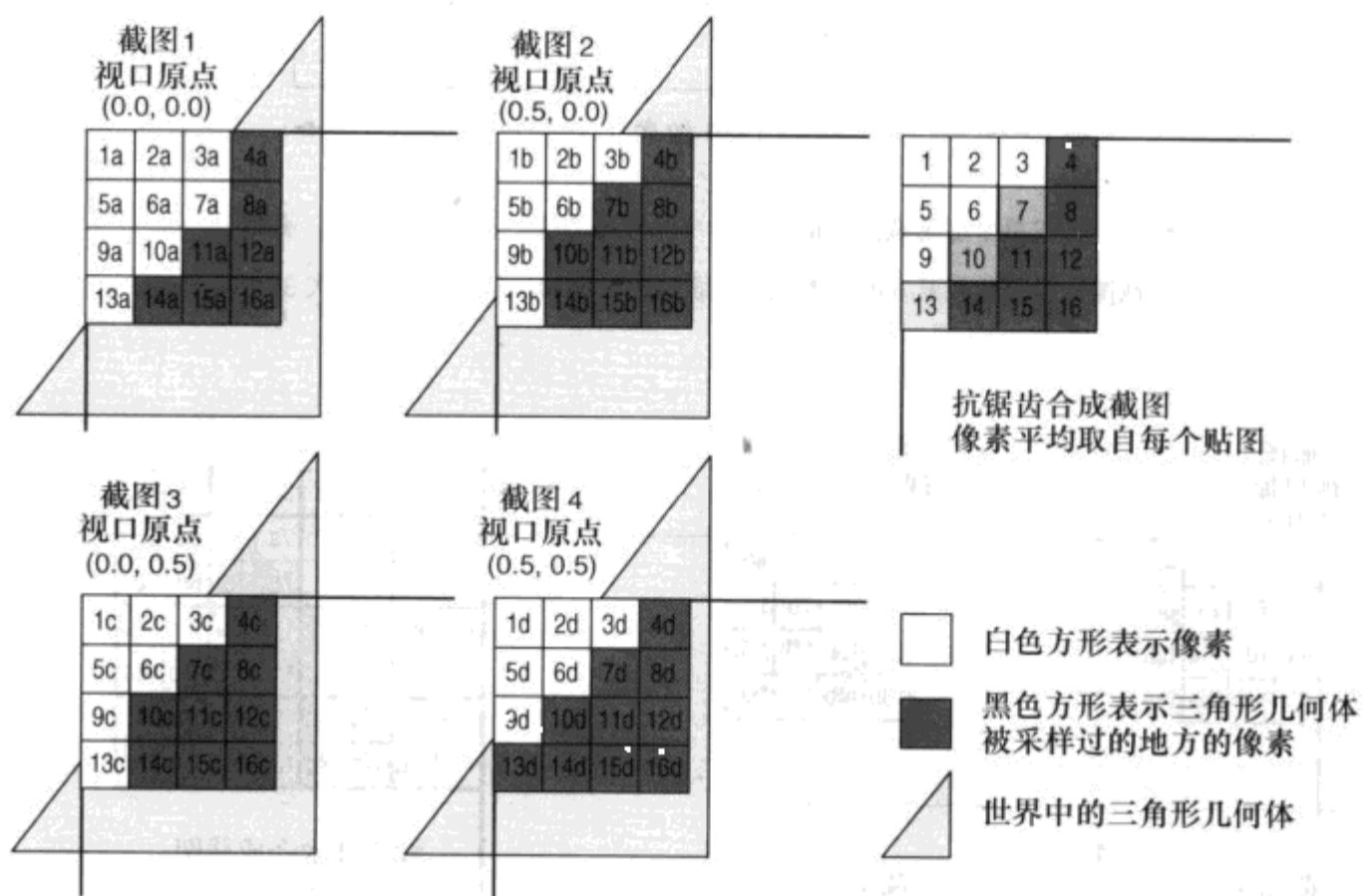


图 5.1.3 将 4 个源屏幕截图被合成，产生一个具有逐像素 4 个采样的抗锯齿截图。

给定编号的像素（比如，像素 13a，像素 13b，像素 13c，像素 13d）被均匀混合到最终截图中的一个单一的像素中（像素 13）

虽然图 5.1.3 所示的技术非常强大，但是当样本都统一地排列在网格中时，抗锯齿就不是很理想。由于网格采样，高频噪音能够逐个透过样本间的缝隙，导致由于细节丢失而产生锯齿。图 5.1.4 显示了一个包含高频细节的单一像素。然而，当网格采样发生的时候，细节丢失了，这是个非常不确切的说法，确切的说法或许应该是，高频细节不能够在单一像素中被捕获，如图 5.14 所示。因此，我们提出了将高频细节转换成噪音的解决方案，平均后的噪音代表像素中的实际细节。三种可实现的随机采样分布如下：随机的(random)、抖动的(jittered)，

和磁盘均衡 (Poisson disc)。

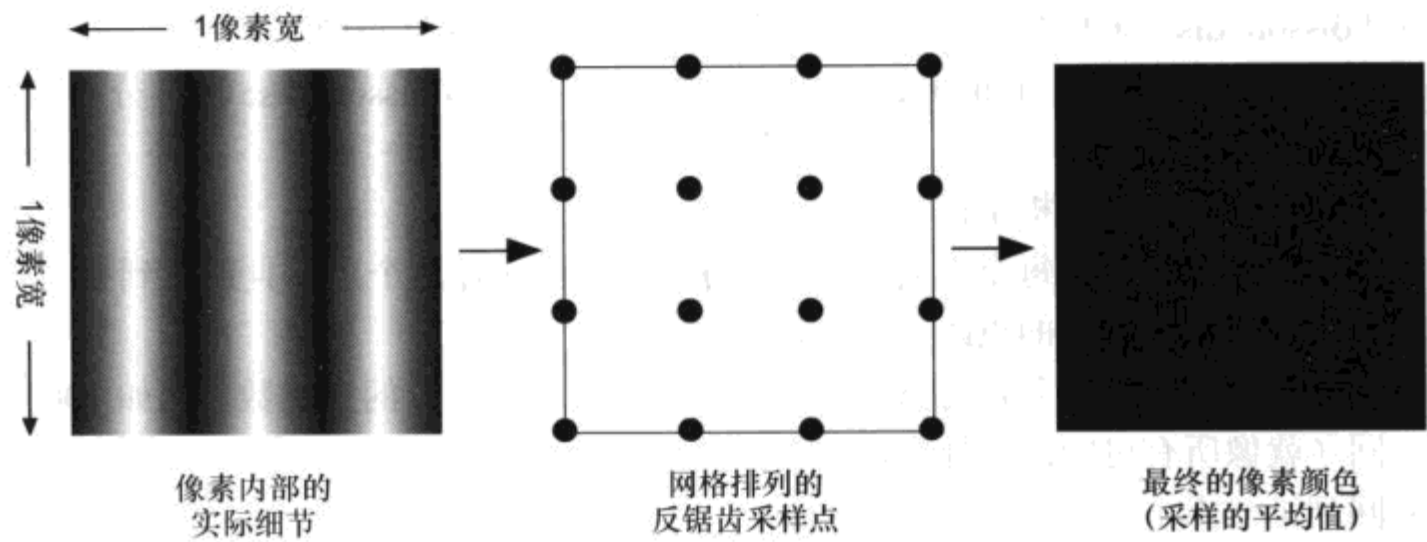


图 5.1.4 具有高频细节的像素在网格上被统一采样。细节透过采样的缝隙，导致像素的误解（这导致许多像素产生锯齿）

1. Random 采样分布

最简单的解决方案就是在像素中随机选择一个样本点，如图 5.1.5 (B) 所示。这看起来好像是合理的，但是它通常会在结果图中产生太多的噪音。而且，往往是当样本采集结束的时候，还有一大片区域未被采样。这是 3 种方案里面最差的一种。

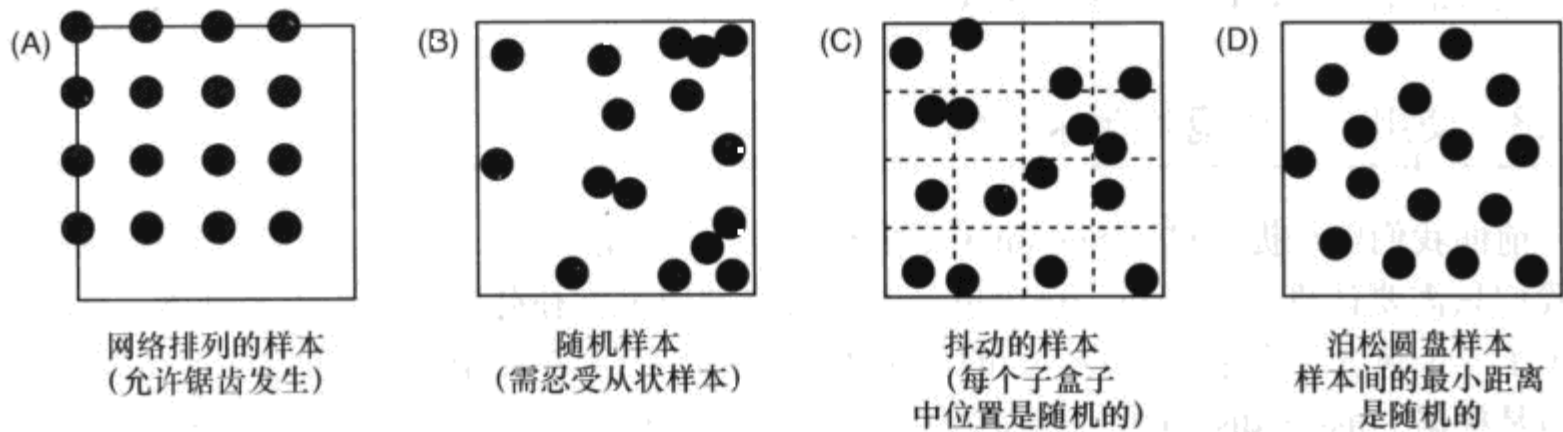


图 5.1.5 采样一个像素的四种可能分布。(A) 代表统一采样。
(B, C, D) 代表不同的随机采样方法

2. Jittered 采样分布

光线跟踪算法一般使用一种 jittering 技术来选择采样点。图 5.1.5 (C) 显示了 jittering 的一个实例。这种技术将像素按网格进行划分，然后在每个网格单元中选择一个随机位置。这可以减少采样点堆聚，并且确保不会有大面积的区域未获得采样。如果你使用得好，这个采样分布就比较接近于理想的解决方案了。

3. Poisson Disc 采样分布

选择子像素样本的理想方案就是在保证样本之间维持一个最小的间距的前提下，可以随便选择位置，如图 5.1.5 (D) 所示。有意思的是，这是解决采样问题的最自然的方式，因为

在某种意义上，就像是在你眼中安置了一个感光器[Yellott82]。其结果是，你的眼睛看不到锯齿了，那些你不能够完全看到的细节变成了细微的噪音或者模糊。

虽然 Poisson disc 采样可以产生最好的结果，但是它的计算花销非常昂贵。这也就是为什么光线跟踪程序倾向于使用 jittered 算法的原因。产生一个 Poisson disc 采样分布的算法主要有以下几步。

- (1) 像素中选择一个随机采样点。
- (2) 如果这个点跟其他每个点的距离都不小于 d ，就保存这个点并且返回到第一步。
- (3) 如果这个点跟其他的点距离太近，就丢弃它。
- (4) 如果这是一行中被丢弃的第 N 个点 (N 是个大于等于 0 的自然数，比如 100)，就返回（就像所有可能的采样点都已经被产生了一样）。
- (5) 回到第一步。

就像你从算法中看到的一样，Poisson disc 采样分布需要花费的时间是 $O(n^2)$ （因为每个已经产生的点必须同其他每个点进行比较），而且我们并不清楚有多少的点将被填充到所给出的空间中。当产生一个分布的时候，你必须指定一个最小的距离 (d)，但是你不能指定实际上产生多少个点。只有反复试验不同的 d 值，才能使你感觉出可以产生多少个采样点。例如，给出一个宽度为 1.0 的像素，如果要求最小的距离为 0.08，那么大概就可以得到 100 个采样点。

有意思的是，术语 Poisson disc 来自于像一个磁盘的磁盘均衡(Poisson disc)分布的 Fourier 变换公式。这一点很重要，因为这些分布的 Fourier 变换公式让我们可以获悉被采样频率的高低。更多的细节和讨论，请参考[Watt92] [Watt99]。

5.1.3 使用一个磁盘均衡采样分布

前面我们曾经提到过，Poisson disc 采样分布计算非常昂贵。然而，这是可以接受的，因为我们只需要计算一次分布。这个单一的分布将被用来弥补截取的每张屏幕截图。图 5.1.6 显示了一个包含三个采样点的单一的分布如何用来弥补三张原始屏幕截图的情况。除了每个视口是根据 Poisson disc 采样点被切换以外，其他他都跟图 5.1.3 很相似，图 5.1.3 中是根据统一的网格切换视口。

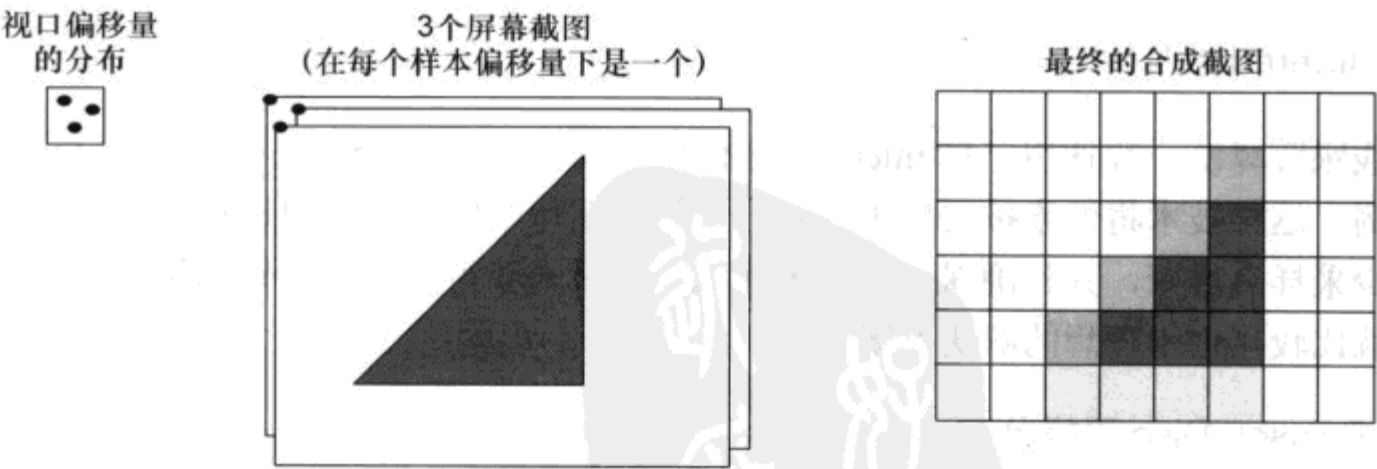


图 5.1.6 将三个源屏幕截图合成，产生一个具有逐像素三个样本的抗锯齿截图。
每个屏幕截图的视口已经根据各自在泊松圆盘分布中相应的样本点被转换过了

5.1.4 为抗锯齿调整像素的采样宽度

为了获得较好外观的像素，另一个可改善的地方就是增加像素采样的面积，甚至可以延伸到相邻的像素中。取样每超过像素边界一位，锯齿就可以进一步减少一些。然而，这个操作必须很小心地调谐才行，因为如果采样面积太深入到相邻的像素的话，将导致图像模糊。实际上，采样宽度为 1.3 的像素效果很好。图 5.1.7 显示了对于给定的 Poisson disc 分布，通过缩放已产生的分布，可以简单地增加或者减少多少的像素采样宽度。图 5.1.8 显示了在特定的范围内通过改变像素宽度所得到的真实结果。

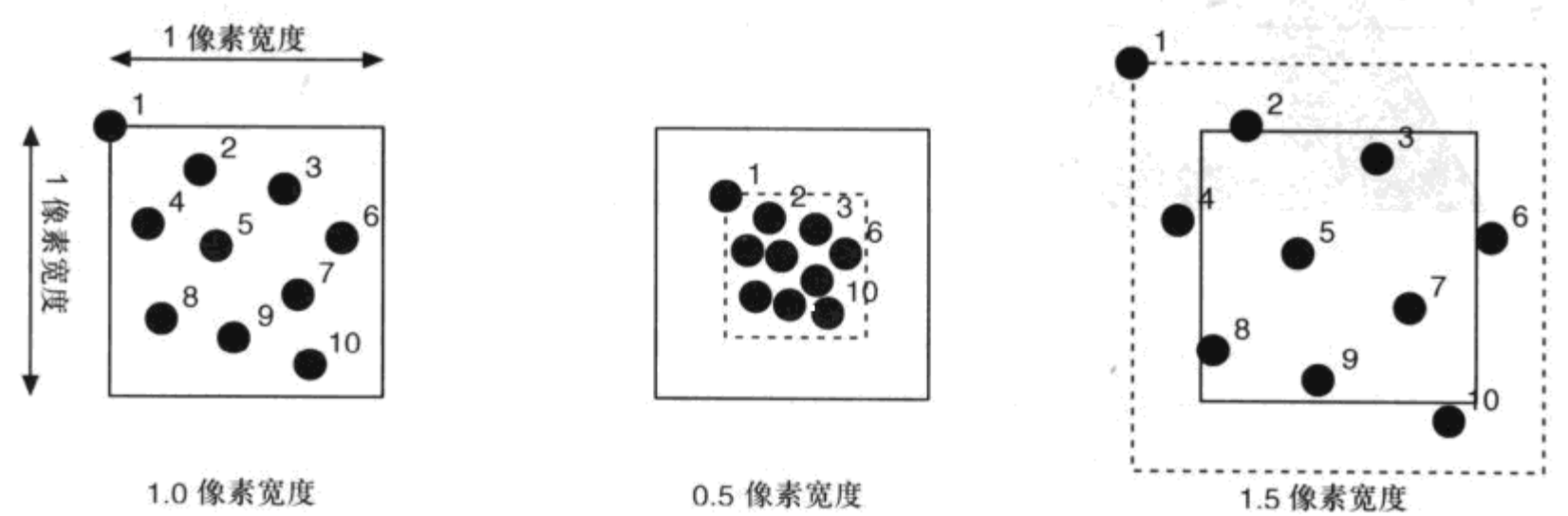


图 5.1.7 通过缩放泊松圆盘分布得到不同的采样宽度

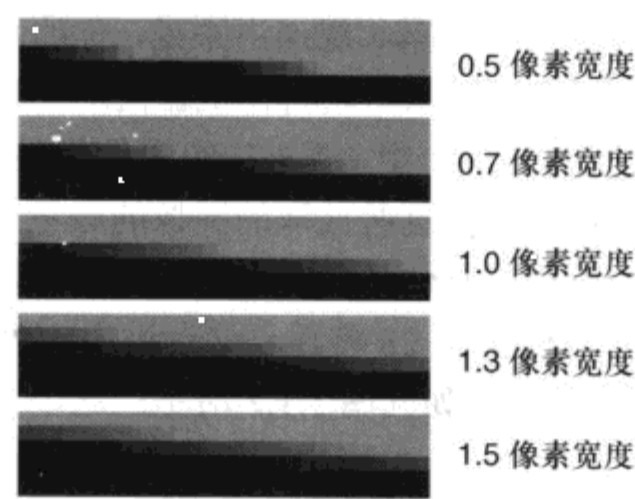


图 5.1.8 不同像素采样宽度的抗锯齿效果。小的宽度的结果是图像清晰，但有锯齿。大的宽度可以消除锯齿，但是图像看起来模糊。在实际工作中，游戏屏幕截图使用 1.3 像素宽度可以达到最佳效果，不过这只是个人观点

5.1.5 增加分辨率同增加像素质量相结合

前一部分已经描述了如何获得一个更大的并且是高度抗锯齿的屏幕截图，但是最终的目的是可以同时制作两个。图 5.1.9 显示了如何创建逐像素 7 个采样点的 4 个抗锯齿像素的整个过程，其中一个像素被取样 28 次。在图 5.1.9 中，28 个采样点中的每一个点都代表了相应贴图（同图 5.1.6 相似）的一个视口偏移量。最后的图像由这些 28 个屏幕截图构成，它们已经被均匀分布并接合到了一起，这个图像有原先的 4 倍大，而且无锯齿。

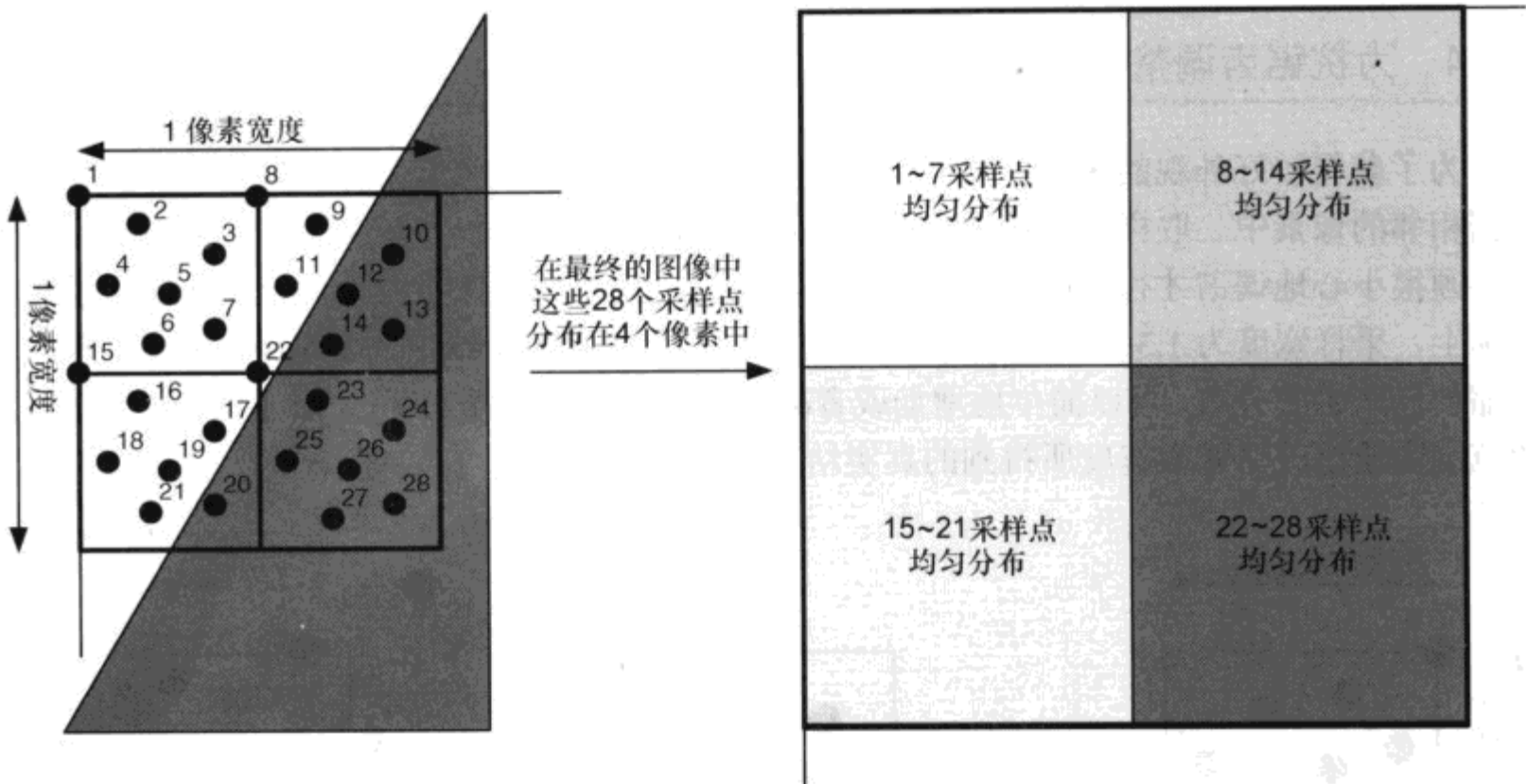


图 5.1.9 一个单一像素被采样 28 次后创建了 4 个像素，每个像素都有 7 个采样点，并且都是抗锯齿的。
每个点表示每个截图的视口原点偏移量。7 个采样点的泊松圆盘分布被产生并在每个像素象限中被使用。
最终的屏幕截图将是原始尺寸的 4 倍，并且是用 7 个样本逐像素抗锯齿的

不幸地是，如果你不小心，复合屏幕截图很快地就能用光所有有效的内存，尤其是在视频游戏控制台（video game consoles）上。收集屏幕截图最好的方法就是备一个额外的帧缓冲（在可访问的 CPU 存储中），对每个红、蓝、绿通道来说，你的本地屏幕尺寸的大小都属于浮点类型。这个特殊的帧缓冲有更多的位深，它将在抗锯齿那步期间被用于积聚几打，成百上千，甚至成千上万的图像。在图 5.1.9 中，1~7 的图像将被截取并放入到特殊的帧缓冲（framebuffer）中。然后，帧缓冲将根据这些截取的图像的数量划分成几块，并为其中的每个像素获得一个平均像素值。在游戏控制台，那些被平均后的帧缓冲将随后被发送到 PC 中临时存储起来。图像 8~14，15~21，22~28 将重复这个过程。然后，在 PC 上，4 张已经被存储的图像将被缝合（thread）到一起，如同图 5.1.2 所示。这个方法确保只使用最少的控制台存储。

下面的伪代码算法描述了在逐像素 7 个样本时，得到 4 倍于原始屏幕截图大的屏幕截图的步骤（例如，从 640×480 到 1280×960），如图 5.1.9 中所示：

```
int group; //表示每组将被均匀混合
           //用来反锯齿的图像
           //在图 5.1.9 中，有 4 组这样的图像

int size_increase = 4; // 产生 4 倍大的屏幕截图
int samples_per_pixel = 7; // 在最终的截图中为 7 个样本逐像素

// 分配基于浮点数的帧缓冲
// 使本地游戏分辨率为 640×480
Fbuffer* fb = new Fbuffer(640,480);
// 逐组循环
```

```
for(group=0; group<size_increase; group++)
{
    fb->Zero(); // 将帧缓冲清 0

    // 一个组中的每个截图进行循环
    for(int cur=0; cur<samples_per_pixel; cur++)
    {
        // 计算视口偏移量, 如图 5.1.9 中的数字标示
        int offset = group * samples_per_pixel + cur;

        // 为视口设置正确的偏移量
        SetViewportToSubPixelOffset(offset);

        RenderShot(); //渲染分辨率为 640×480 的截图
        AddShotIntoFramebuffer(fb);
    }

    // 通过截取截图的数量来划分帧缓冲(本例中为 7)
    // 结果图是一组反锯齿的截图
    fb->Average(samples_per_pixel);

    // 将基于浮点的帧缓冲转换成基于字节的
    fb->ConvertToByteBased();

    // 把基于字节的帧缓冲数据从游戏机
    // 发送到其所连接的一个 PC
    SendToPC(fb);
}

// 所有的截图被截取
// 使用图 5.1.2 中的线程技术将已连接的
// PC 上的图像合成起来
```

就像前面算法中显示的那样, 在游戏控制台上, 抗锯齿那一步并不需要太多的内存(大约 3.5MB), 并且因为多数贴图被积聚在特殊的基于浮点的帧缓冲中, 所以完全可以测量出来。即使截取逐像素 1000 个样本的贴图, 控制台内存的需求量还是保持不变。然而, 考虑到最终图像的尺寸, 合成的那一步(在 PC 连接到游戏控制台上时执行)仍然需要大量的内存。例如: 一个 640×480 的屏幕截图扩大 625 倍, 就是 16 000×12 000。一个如此大小的屏幕截图将需要大约 550MB ($16\,000 \times 12\,000 \times 3\text{Byte} = 549.32\text{MB}$) 的内存, 因此在合成的那一步, 产生的巨大贴图的内存需求可能会达到你 PC 的极限。

5.1.6 结论

这篇文章阐述了如何改进原始的屏幕截图, 使它们变得更大, 更漂亮。关键在于截取大量的屏幕截图, 按照 Poisson disc 分布, 每一张截图都用像素的一小部分作为自己的视口(viewport)偏移量。通过平均和合成, 成千上万的屏幕截图被合成为一张单一的货真价实的高分辨率贴图。

这项技术已经成功地用来产生过 $19\,200 \times 14\,400$ 大小的贴图，在逐像素 100 样本，硬件采用 Nintendo GameCube 的条件下，产生这么大的贴图需要 90 000 张单独的屏幕截图，大概需要花费 1 个小时来捕获（25 分钟用于渲染，20 分钟用于传递到 PC 上，15 分钟用于在 PC 上合成）。相反，一张 1920×1440 大小的图像，在逐像素 100 样本的条件下，总共大约只需要花费 25 秒。

无论你所截取的是什么样的贴图，关键是要完全控制尺寸和质量，以让你可以用最好的光线来展示你的游戏，满足你所有的需求。

5.1.7 参考文献

[Vlachos01] Vlachos, A., and E. Hart, "Rendering Print Resolution Screenshots," *Game Programming Gems 2*, Charles River Media, 2001.

[Watt92] Watt, A., and M. Watt, *Advanced Animation and Rendering Techniques: Theory and Practice*, Addison-Wesley Publishing Co., 1992.

[Watt99] Watt, A., *3D Computer Graphics, Third Edition*, Addison-Wesley Publishing Co., 1999.

[Yellot82] Yellot, I., "Spectral Analysis of Spatial Sampling by Photoreceptors: Topological Disorder Prevents Aliasing," *Vision Research*, 22, 1982, pp. 1205–1210.



5.2 非封闭网络模型的 GPU 容积阴影构架

作者: Warrick Buchanan

E-mail: warrick@chimeric.co.uk

译者: 刘永静

审校: 谷超

最近几年以来, 容积阴影[Crow77]已经成为一个提高游戏中灯光质量的通行技术。这是由于在大部分的现代图形硬件中, 有模版缓冲(stencil buffer)可用, 它促进了这项技术的发展[Heidmann91], 并且在过去的几年里, 更高级的图形硬件已经被制造出来, 它们可以为容积阴影的实现提供强力的支持[Kilgard01]。在图形硬件顶点阴影单位上创建容积阴影是当前被广泛推行的技术, 但是它有一些前提条件限制, 就是主题场景应为单边的、全封闭的网格模型。此类网格模型中, 共享每一条边的三角形有且只有两个(一个双面(two-manifold)网格模型)[Brennan02]。

当然, 这并不是主要的问题, 让人满意的是, 对于给出的任何单边网格模型拓扑应用, 这项技术都是正确的。在生产环境中, 如果没有特殊的处理, 封闭的非双面网格模型不能自动创建。这篇文章阐述了一个在不需要封闭的双面网格模型的情况下, 渲染容积阴影的方法。

5.2.1 回到制图板

为了正确地构造一个容积阴影, 我们必须考虑到这样一个事实, 那就是只有那些对光源可见的面才能决定容积的形状, 因为它们是实际阻挡光线的面。

最有效的方法就是通过获取这些平面本身, 朝光源方向突出的反面, 使用原始面和突出面的边缘构造出的面, 简单地为每个对光源可见的平面构造一个容积阴影。通过这些操作我们可以为每个面产生一个相容且封闭的容积阴影。虽然这种方法对所有网格模型都是正确的, 但是, 这显然不是一个可行的方法, 因为对几乎任何应用程序, 填充率都将会是一个很大的麻烦。

虽然这个算法可以工作, 但远非是最有效的, 因为它没有考虑我们所关心的关于网格面连接的信息。在所有的容积阴影技术中, 我们实际上只想要用突出的轮廓边缘构造出容积阴影的边, 这些边介于各个面之间, 它们是有灯光面和无灯光面的分界线。如果我们按照以前的规定, 使用突出的轮廓边缘和前后容积底面(volume cap)来构造出阴影容积边, 那么我们可以使用的技术都只能用于封闭的双面网格模型, 而不能用于未封闭的

网格模型和非双面网格模型。

为了处理打开边缘的网格模型的问题，我们必须扩展轮廓边缘的分类。一个同时也是打开的边缘的轮廓边缘(面上的一条边是不被用来连接相邻面的)，它的关联面是对光源可见的。在这个新的规则下，该技术就可以用于非封闭的双面网格模型了。然而，它还是不能处理那些在面的一个边上不止有一个相邻面的问题(非双面网格模型)。

关于这一点，我们可以通过配对共用边的面来解决。以那些具有被标记为打开的边的面为起点，然后，查找每个面，看是否有其他面同它共享一个边。如果我们所查找的一条打开的边上的面，在这条边上还没有其他面同它配对的话，我们就把它同通过那条边的面进行配对。如果我们所查找的面在那条边上已经跟其他面配对了，我们就不要注册它们之间的连接，而是要把它们分开。这个过程有效地把那些网格模型沿着非双面边分成了很多互不相干的部分，每一部分或者具有双面的连接，或者成为一个轮廓边。经过这样的处理，该技术也可以用于非双面网格模型了。

5.2.2 在顶点阴影中实现这项技术

思考一下上述的修改，我们可以把整个任务分成以下三个途径来完成渲染：产生容积阴影的前底面(front cap)，产生容积阴影的后底面(back cap)，产生容积阴影的边。我们需要3种单独的渲染途径，因为前后底面(front and back cap)虽然使用相同的几何数据，但却使用不同的顶点阴影，这一点跟它们的选择次序一样(别忘了，在本技术中，后底面是通过翻转前底面的缠绕顺序而得的)。最后的第三个途径是必需的，它用来产生阴影容积的边，连同定义四边形的索引缓冲一起，我们同样使用相同的几何数据，不同的顶点阴影。

1. 前底面

前底面是容易产生的。我们只需要丢弃掉所有未朝向光源的三角形就可以了。如果我们已经复制了顶点的数据(每个三角形的三个惟一顶点)，通过每个顶点和它所在平面的法线，我们就可以使用点积确定该顶点是否是朝向光源的三角形的一部分。如果它是，我们就不对它进行任何操作。但是如果它不是，我们就输出该顶点距离原点的位置。结果是，将创建硬件不绘制的退化的、零范围的三角形(degenerate, zero-area triangles)(参看程序清单 5.2.1)。

程序清单 5.2.1 渲染前底面的 Vertex shader

```
; c[0] 0.0, 0.5, 1.0, 2.0
; c[1-4] world*view*projection matrix
; c[5] light position

vs.1.1

dcl_position v0
dcl_normal v1
dcl_texcoord v2

; Get vector from vertex to light and normalize
sub r0,c[5],v0
```

```

dp3 r0.w, r0, r0
rsq r0.w, r0.w
mul r0, r0, r0.w

dp3 r0,r0,v1
mov oD0,r0

; Make r0 (1, 1, 1, 1) if normal is facing light else (0, 0, 0, 0)
sge r0, r0, c[0].xxxx

; Transform position to clip space
dp4 r1.x, v0, c[1]
dp4 r1.y, v0, c[2]
dp4 r1.z, v0, c[3]
dp4 r1.w, v0, c[4]

mul oPos, r0, r1

```

2. 后底面

这里的后底面不同于在其他著名的方法中使用的后底面。它们实际上只是前底面沿着光线的方向向后突出一段距离所形成的。因为我们想要它们担当一个后底面多边形，所以它们需要面向一个同前底面所面向的方向相反的方向。这个突出在顶点阴影中很容易完成，但是当绘制后底面的时候，我们必须记得要倒转三角形的选择顺序。如果我们简单地按照其原来的顺序拿来使用，它们的缠绕次序将会不正确（参看程序清单 5.2.2）。

程序清单 5.2.2 渲染后底面的 Vertex shader

```

; c[0] 0.0, 0.5, 1.0, 2.0
; c[1-4] world*view*projection matrix
; c[5] light position

vs.1.1

dcl_position v0
dcl_normal v1
dcl_texcoord v2

; Get vector from vertex to light and normalize
sub r0, c[5], v0
dp3 r0.w, r0, r0
rsq r0.w, r0.w
mul r0, r0, r0.w

; Dot light and normal vector
dp3 r1, v1, r0

; Output shading
mov oD0,r1

```



```

; Make r2 (1, 1, 1, 1) if normal is facing light else (0, 0, 0, 0)
sge r2, r1, c[0].xxxx

; Extrude vertex
mad r0, c[5].www, -r0, v0

; Transform position to clip space
dp4 r1.x, r0, c[1]
dp4 r1.y, r0, c[2]
dp4 r1.z, r0, c[3]
dp4 r1.w, r0, c[4]

mul oPos, r1, r2

```

3. 凸多边形

容积的边稍微有点问题。我们以使用传统的硬件加速技术，通过在每个三角形的边上构建退化的四边形作为开始[Brennan02]。注意，不需要新的顶点，因为一个索引缓冲完全可以满足定义四边形。而且，在那些共面的三角形之间的边上，不需要四边形。在其他技术中，如果顶点面向光源，我们就不动它，否则，我们应该沿着光源的方向突出它。

在这一点上，我们还有一个算法可以用来解决问题，不过它只对全封闭的网格是正确的。为了处理打开的边缘的问题，我们首先复制边缘的顶点，并且转化每一个新顶点的平面法线。然后，我们可以使用新的顶点和原始边缘的顶点为这个边缘产生一个四边形。把这几步加到我们的算法中，就可以得出一个正确的容积阴影。但是还有一个问题没有解决，就是光线能够看到使用打开的边的三角形的背面。

为了修正这个问题，我们必须确保不可以挤压源自朝后（back-facing）的多边形的一条打开的边的轮廓边缘的四边形。为了做到这一点，我们需要能够标记我们已经增加的属于打开的边缘（open edges）的顶点。众所周知，每个顶点法线可近似地看作单位长度，因此我们可以用法线矢量标志这些打开边缘的复制点，它们的长度只比某个数（在某个 epsilon 范围之内）略大。

这个不影响我们对面向光线的三角形/顶点的测试，而且它意味着我们不需要为每个顶点使用一个标志来存储额外的部分。因此，在产生容积阴影的时候，如果我们已经有了被标记为打开边（open-edge）的复制顶点的顶点的时候，我们常常使它们突出来。如果他们未被标记，我们就只使那些法线背离光线的面突出来。现在，当我们在先前那个导致问题发生的案例中再去创建不可见的退化的容积阴影边时，此算法可以应用于所有的网格拓扑中，并将产生正确的行为（参看程序清单 5.2.3）。

程序清单 5.2.3 渲染凸边的 vertex shader

```

; c[0] 0.0, 0.5, 1.0, 2.0
; c[1-4] world*view*projection matrix
; c[5] light position

vs.1.1

```

```

dcl_position v0
dcl_normal v1
dcl_texcoord v2

; Get vector from vertex to light and normalize
sub r0,c[5],v0
dp3 r0.w, r0, r0
rsq r0.w, r0.w
mul r0, r0, r0.w

; Dot light and normal vector
dp3 r1,v1,r0

; Output shading
mov oD0,r1

; Make r1 (0, 0, 0, 0) if normal is facing light else (1, 1, 1, 1)
slt r1, r1, c[0].xxxx

; Make r2 (1, 1, 1, 1) if normal is not unit length else (0, 0, 0, 0)
mov r2, v1
dp3 r2, r2, r2
sge r2, r2, c[0].w

add r1, r1, r2
min r1, c[0].zzzz, r1

; Extrude vertex if facing away from light
mul r0, r0, r1
mad r0, c[5].www, -r0, v0

; Transform position to clip space
dp4 oPos.x, r0, c[1]
dp4 oPos.y, r0, c[2]
dp4 oPos.z, r0, c[3]
dp4 oPos.w, r0, c[4]

```

5.2.3 需要注意的事项

如果需要，我们可以忽略掉容积的个体部分以使容积阴影进一步优化，因为容积的三个部分（前底面、后底面、边）是单独提交的。而且，如果知道我们的模型是封闭的，并且每条边有且只有两个三角形，就可以使用代价不高的依赖于封闭双面网格模型传统技术。

在最初的顶点阴影生产技术中，我们需要复制每个面的每个顶点，而且，必须为每条打开的边增加额外的一对顶点。这就导致了顶点数据的大幅度增长，而且这些数据必须被发送到图形卡。由这个特殊的技术导致的另外一个不受欢迎的结果是，为了正确地构建容积的前底面、后底面和边，这些几何（geometry）必须被提交三次。

近来, 硬件已经实现了双面模版, 可以用来测试加速容积阴影渲染。这仍然可以连同这项技术一起使用, 然而, 由于翻转它们的缠绕顺序的需要, 后底面必须使用反向的双面模版逻辑来完成渲染。因此, 如果你在绘制容积的前底面和边时, 将容积阴影面前面的 stencil count (模版数量) 设置为逐步递增, 将后面的 stencil count 设置为逐步递减, 那么你在绘制容积后底面的时候, 就必须交换递增和递减的操作, 即将前面的 stencil count 设置为逐步递减, 将后面的 stencil count 设置为逐步递增。

本文介绍了只需要依靠面向光线的面就可以产生容积阴影的技术, 它不同于当前的其他技术。不过, 这些打开的和非双面的边必须被正确地标示出来。

5.2.4 结论

从这篇文章中我们可以知道, 开发人员可以为美工提供更广泛的生产流程, 并且可以充分利用现代图形卡提供的加速性能。当三角形数量越来越少的今天, 把产生容积阴影的更多前提工作提交给图形卡实现已经变得越来越至关重要。

5.2.5 参考文献

[Brennan02] Brennan, Chris, "Shadow Volume Extrusion Using a Vertex Shader," in Engel, Wolfgang, ed., *ShaderX*, Wordware, May 2002.

[Crow77] Crow, Frank, "Shadow Algorithms for Computer Graphics," *Proceedings of SIGGRAPH 1977*, pp. 242-248.

[Heidmann91] Heidmann, Tim, "Real Shadows Real Time," *IRIS Universe*, Number 18, 1991, pp. 28-31.

[Kilgard01] Kilgard, Mark, "Robust Stencil Volumes," CEDEC 2001 presentation, Tokyo, September 4, 2001.



5.3 透视阴影贴图

作者: Marc Stamminger, University of Erlangen-Nuremberg

George Drettakis, REVES/INRIA Sophia-Antipolis

Carsten Dachsbacher, University of Erlangen-Nuremberg

E-mail: stamminger@cs.fau.de,

George.Drettakis@sophia.inria.fr,

dachsbacher@cs.fau.de

译者: 刘永静

审校: 谷超

阴影贴图是最流行的阴影生成算法之一。阴影贴图是光源视图的深度缓冲 [Williams78] [Reeves87]。它的具体做法就是, 将阴影贴图中的一个点的阴影对一个作为阴影的贴图的点进行测试, 并且做一个简单的深度比较。阴影贴图效率高, 非常通用, 相当健壮, 容易实现, 并且被当前图形硬件所支持。它的主要缺点是会产生阴影锯齿, 这是由像素 (离散的) 阴影贴图所引起的。这种现象在大场景中尤其显著, 因为这时候阴影贴图必须覆盖大片的区域, 因此在前景区通常会产生太低的分辨率。

透视阴影贴图 (PSM) 与标准阴影贴图非常类似, 但是已经经过了变形。因此离摄像机近的区域比远的区域的分辨率更高 [Stamminger02]。之所以称为“透视”是因为它们说明了观察者的透视变形。在阴影贴图上的物体大小同它们在最终图像中的大小相符合。在理想情况下, 有限大小的 PSM 能在无限大小的场景中生成阴影 (例如, 过程地形), 并且没有明显的阴影像素。

5.3.1 引言

透视阴影贴图 (perspective shadow map) 的概念放在后透视 (post-perspective) 空间中来想象很容易理解。换句话说, 在这样的空间里其模型视图 (modelview) 和投影矩阵 (projection matrix) 已经被应用。在这样的空间里, x 和 y 坐标就是在图像中的最终位置, z 坐标是深度值。在后透视空间, 透视变形 (perspective distortion) 已经被应用; 就是说, 靠近摄像机的物体被放大, 远离摄像机的物体被收缩。在这样的后透视空间中, PSM 包含场景光源视图。因此, 离观察者近的区域比远的区域占有更高的世界空间分辨率, 导致在图像的密度抽样和阴影贴图之间有更多的一致性。

如图 5.3.1 中的例子, 它显示了在地平面上一个简单的 2D 场景, 它有三个角色。平行光源从上面照亮了场景。在一幅标准阴影贴图中, 场景的

渲染是通过沿着 z 轴的平行投影方向进行（图 5.3.1 左）。而在后透视空间（图 5.3.1 右）中，观察者的位置场景被变形了，最近的角色被放大了。在这后透视空间里，透视阴影图是一个光源视图，因此最近的角色比那些远的角色获得更高的阴影贴图分辨率。图像也显示了分辨率的匹配：如果我们将阴影贴图的单个像素投射到地面，然后再投到图像平面（image plane），那么所有阴影贴图像素会覆盖相同的图像区域。

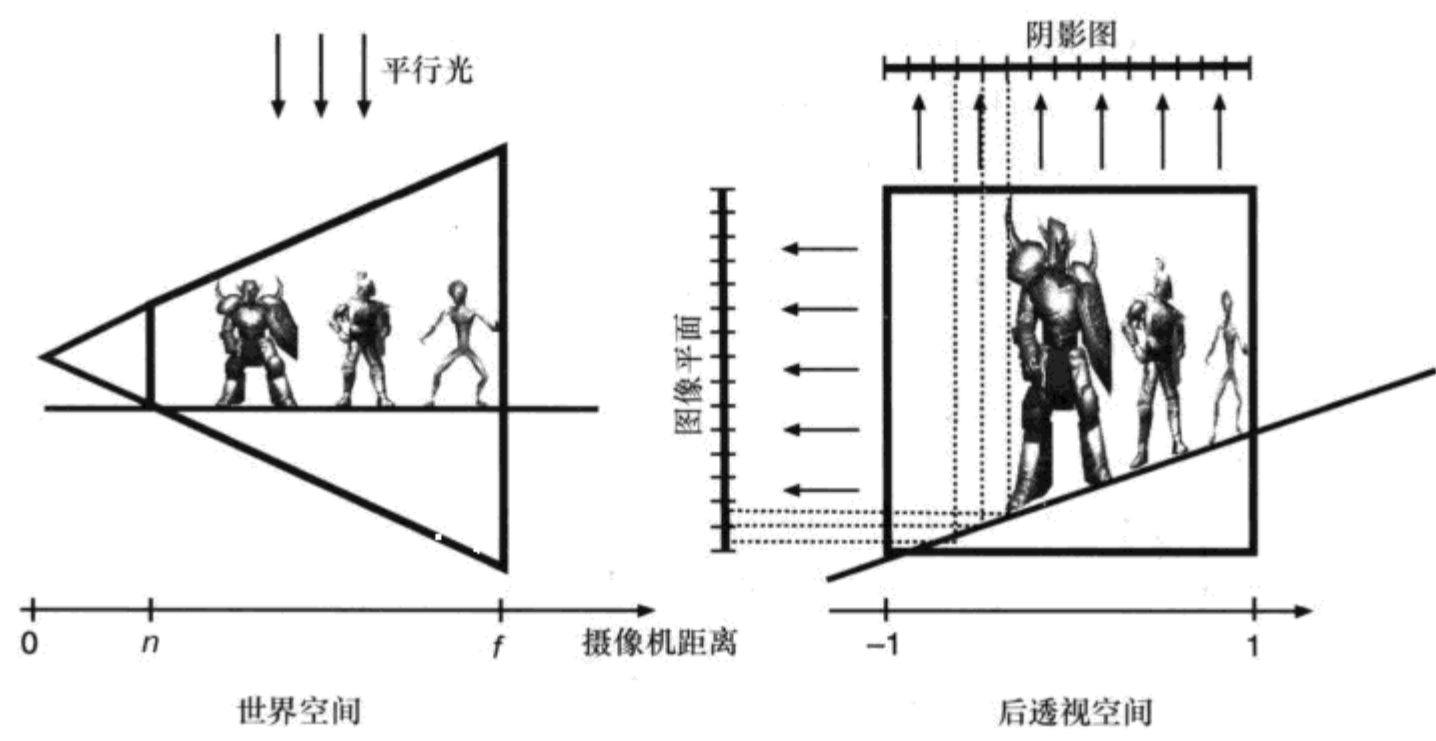


图 5.3.1 世界空间中的场景（左边）和它在后透视空间中使用透视阴影贴图的副本

前面的例子是特意选取的说明 PSM 工作的最好情形的一个例子。对平行光源来说，如果光的方向与摄像机平面平行，并且阴影平面与光照方向垂直，那么 PSM 就是理想的。如果图 5.3.1 中的光源被移动到前面（朝向观察者发光），阴影就被投射向观察者。由于阴影更靠近观察者，因此比遮光板（occluder）更大，结果导致像素化（锯齿）。如果光源从摄像机后面发光，在摄像机平截面后的物体就会被包括进来。这些需要被特殊处理，而这也负面地影响了阴影分辨率。如果光源被移到一边，那么阴影就会沿着地平面伸展，并因此被放大，这样就会暴露阴影贴图的像素结构。

PSM 是为大场景开发的，通常为室外场景，因为在这种情形下标准阴影贴图失效了。由于大场景通常被一束平行光源（太阳）照亮，在这篇文章中我们仅处理平行光。在阴影贴图设备环境中，点光源通常不是全方向性的，只是光斑，它能被一个单一的阴影贴图很好地处理。它们一般显示出距离衰减，所以它们有一个影响的限制范围，因此就不能从透视阴影贴图受益很多。

在下一节中，我们将第一次讨论在后透视空间中的场景的性质。那时我们会演示怎样生成 PSM 和如何避免典型的缺陷。为了实现结果我们考虑使用 OpenGL，DriectX 的实现也是非常相似的。DirectX 中的实现的主要不同点是 API 的使用，透视变换贴图深度的范围为 $0 \sim 1$ ，而不是在 OpenGL 中的 $-1 \sim 1$ 。

5.3.2 后透视空间

既然 PSM 存在于后透视空间，那么将其应用在计算机图形学方面对理解透视的概念是

必要的。透视变换是用一个 4×4 变换矩阵描述的。和通常的组成平移、旋转、比例、剪切的仿射模型 (affine modeling) 变换形成对比, 透视转换运用于矩阵中的所有条目, 因此也能创建投影变换。我们假定读者非常熟悉齐次坐标和矩阵的基本概念。

投影变换映射一条条的直线。然而, 和仿射变换形成对比, 在射影变换中, 平行线不再保持平行。很明显, 这是透视投影的需要, 在世界空间中轨道是平行的, 但是在图像中轨道是聚合的。另一种解释是, 投影变换能映射无限远处的点到限定的点, 反之亦然, 因此两平行线的无限相交点被简单地映射到两非平行线的限定相交点上。

透视变换通过一个 `gluPerspective()` 调用生成, 它能生成一个投影变换, 可以映射场景到一个我们称之为后透视空间的空间。在这样的空间中, 场景是变形的, 因此它有如下性质。

- 观察者被移到无限远的地方 $(0, 0, -\infty)$, 因此通过观察者所有直线变成了与 z 轴平行的直线。
- 摄像机锥体被映射到单位立方体 $[-1, 1]^3$, 近平面被映射到 $z = -1$ 的平面, 而远平面则被映射到 $z = 1$ 的平面。
- 在世界空间中无限远处的点被映射到 $z = z_\infty = (f + n) / (f - n)$, f 和 n 分别是远平面和近平面的距离。在正常的设定里, f 是大的, z_∞ 只比 1 稍为大一点, 我们称 $z = z_\infty$ 为无限平面 (infinity plane) (见图 5.3.2)。

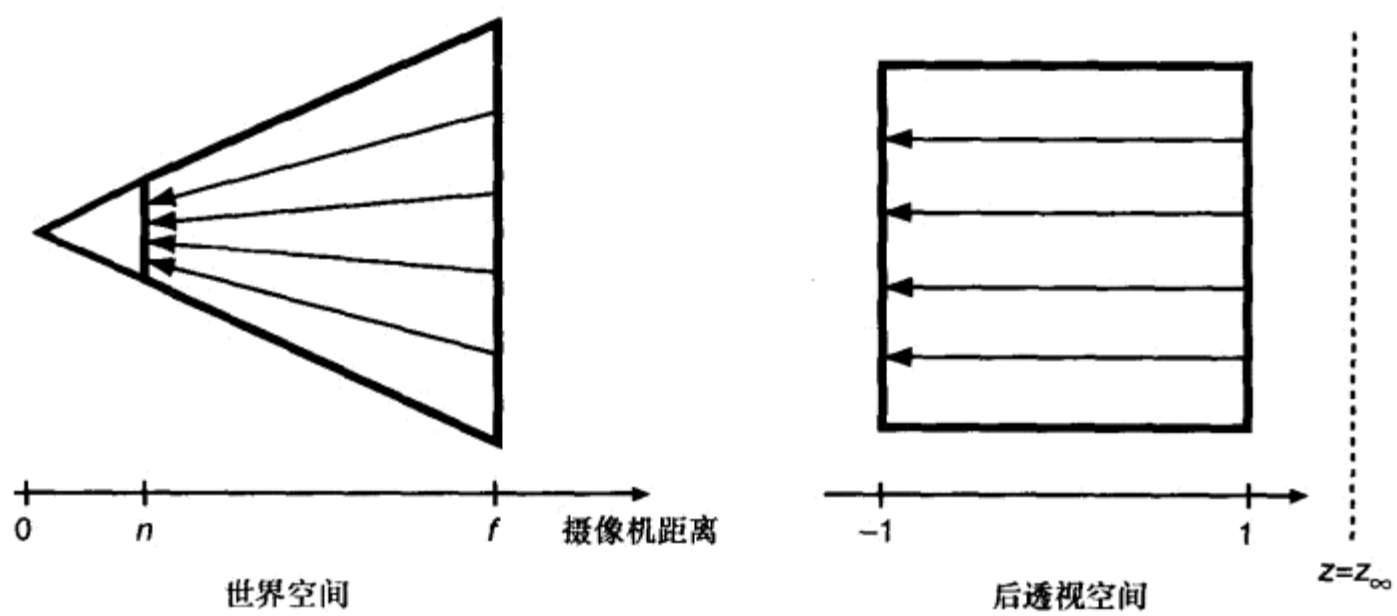


图 5.3.2 世界空间 (左图) 中和后透视空间 (右图) 中的视锥体和投射光线

通过后透视场景的一个简单平行投射到单位立方体的前侧面可得到最终的图像。这意味着在后透视空间一个点的 x 和 y 是这个点的图像坐标, 而 z 则对应为深度。然而, 要注意的是, 后透视空间的深度并不是世界空间深度的线性变化。

从某种形式上, 我们可以这样说, 透视变换挤压了场景, 使得远处的物体变得小了而靠近观察者的物体则被放大了 (见图 5.3.1)。当摄像机移动的时候, 缩放比例也跟着变化, 渐渐靠近观察者的物体被放大, 而渐渐远离观察者的物体则被缩小。后透视空间的一个非常好的特性就是物体的大小与在最终图像中一样。这是我们将要为 PSM 使用的关键特性。

5.3.3 后透视空间中的光

透视变换对光源有令人惊讶的影响。光源是一束全都与平行光平行，或者是源自点光源的有限原点的光子射线。由于光束的性质是通过投射映射到后透视空间，因此光源的类型可以改变。因此一般来说，在世界空间中的一束平行光被映射为后透视空间中的点光源。接下来，我们要总结一下在世界空间中的平行光到其在前透视空间中的匹配物对应的转换。

在世界空间中，平行光源是有无限远的原点的点光源。现在，到后透视空间的投影变换能够映射这无限远的原点到一个限定的原点，因此光源被映射成后透视空间中的一个点光源。同样，在世界空间中的一个点光源（限定的原点）能被变换成后透视空间中的平行光（无限远处之原点）。在这篇文章中，我们限定为世界空间中的平行光源，因为它们是 PSM 最有意思的应用领域。

正如先前所提到的，透视变换映射无限远处的点到有限平面 $z = z_\infty = (f+n)/(f-n)$ 的点。因此，通常一个平行光源（原点在无限远处）变成后透视空间（原点在无穷大的平面上）的一个点。为了更彻底地分析它，我们给出了如下的例子，它在图 5.3.3 中也有描述。

- 一垂直于观察方向的平行光源在后透视空间中的 xy 平面中仍然是平行光（图 5.3.3 左）。
- 一个朝观察者照射的平行光在后透视空间中的无穷大平面上成为一个点光源（图 5.3.3 中）。
- 一束来自观察者后面的平行光变成后透视空间中的无限平面下面的一个点光。来自原始平行光源的所有光线被映射成会聚于一点的光线。这意味着光并不是从无穷大平面上的点到场景传播，而是仅仅朝向无穷大平面下面的光传播。这样的光源不是非常直观，因为它们没有真实的物理对应物，但事实上它们能被反向深度（reversing depth）（图 5.3.3 右）容易地处理。

一个平行光源的世界空间原点在无穷远处。如果光从 $(d_x, d_y, d_z)^T$ 这个方向照射过来，在齐次坐标中，它的原点可以用点 $(d_x, d_y, d_z, 0)^T$ 表示。如果 P 是世界空间到后透视空间的变换矩阵，那么我们就可以计算后透视光原点 $p = (p_x, p_y, p_z, p_w)^T = P(d_x, d_y, d_z, 0)^T$ 。如果 $p_w = 0$ ，那么表示后透视原点在无穷远处，换言之，光源在后透视空间中也是一束平行光（例子见图 5.3.3）。只有当 d 垂直于观察方向（viewing direction）时才发生这样的情况，在本问题中， p_z 永远是 0；换言之，后透视光的方向是 $(p_x, p_y, 0)$ 。如果 $p_w \neq 0$ ，那么就表示后透视光在限定的位置 $(p_x/p_w, p_y/p_w, p_z/p_w)$ ，我们知道 $p_z/p_w = z_\infty$ 。然后我们必须判定是有一束光源还是光汇点（light sink）（情况 b 或者 c）。一个可能的判断依据是观察方向和光照方向之间点乘积的符号，它告诉我们光是从后面还是前面照射过来的。如果从 `gluPerspective()` 获得的标准 OpenGL 射影矩阵被使用，那么我们也可以看看 p_w 的符号：如果 $p_w > 0$ ，那么我们有一个光源（情况 b）；如果 $p_w < 0$ ，那么我们有一个光汇点（情况 c）。

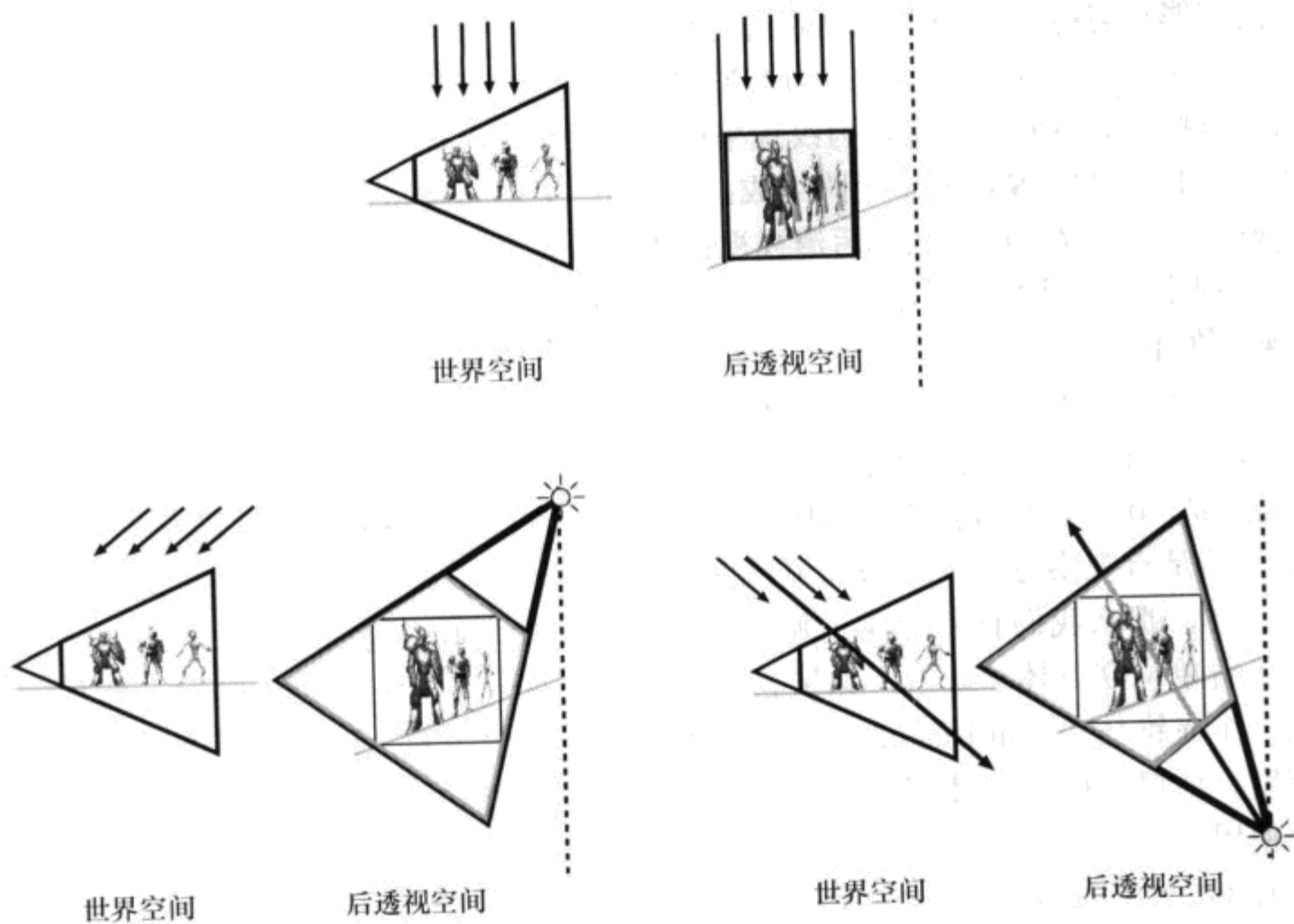


图 5.3.3 光源转换：与视觉方向垂直的平行光在后透视空间中也是平行的（情形 a，左边的图）。世界空间中从观察者前面而来的平行光源在后透视空间中变成了在无限的平面上的一个点光源（情形 b，中间的图）。来自观察者后方的平行光在后透视空间中变成了沉在无限的平面下的一个点光（情形 c，右边的图）。注意在最右边的情形中沿着光线的物体的次序是如何变化的

5.3.4 透视阴影贴图

标准阴影贴图是通过从光源视图渲染场景来创建的。对聚光灯来说，摄像机被放到光源的位置，它的观察方向和可视区被设置成光点的方向和可视区。对一平行光源来说，观察方向与光平行的一个直角摄像机被选中。

与此相反，透视阴影贴图使用后透视光源在后透视空间中生成。场景和光源是第一次被变换到后透视空间。然后，我们为后透视光源生成阴影贴图，就是在后透视空间中看到的场景。

在后透视空间中，靠近观察者的物体被放大，远离观察者的物体被缩小。因此在透视阴影贴图中，靠近观察者的物体可获得比远离观察者的物体更高的分辨率。如果我们假定近的阴影来自近的物体而远的阴影来自远的物体，那么我们将看到近的阴影受益于渐渐增强的阴影贴图分辨率，而远的阴影区域则降低了阴影贴图分辨率。

1. 构造

构造一个透视阴影贴图意味着构造一个矩阵 S ，这个矩阵把在世界空间的一点映射到阴影贴图空间。在映射后， x 和 y 是阴影贴图的纹理坐标，而 z 是光源的深度值。只要沿着光线的所有点被映射到一个惟一的 (x, y) 位置，任何矩阵 S 都是有效的。由于一个点和它所

有可能的阴影投射者 (caster) 被映射到阴影贴图中的一个单一点, 因此阴影测试变成一个简单的深度比较。矩阵 S 用到两次: 当渲染阴影贴图的时候, 它作为射影矩阵被第一次使用; 当应用阴影贴图的时候, 它又作为纹理坐标矩阵被使用。

在一个 PSM 里, S 由两个矩阵组成: $S = CP$ 。 P 将点从世界空间映射到后透视空间, 它由当前观察者所定义。 C 将点从后透视视锥体 (view frustum) (单位立方体) 映射到 PSM。 C 由后透视空间的视锥体所定义, 它有后透视光源作为原点并紧紧围绕在单位立方体周围 (后透视光锥体见图 5.3.3)。

为了计算 C , 我们首先变换平行光到后透视空间, 判定 a 、 b 或者 c 三种情形, 从而生成 C 。因此, 我们首先变换世界空间平行光的原点到后透视空间: $p = (p_x, p_y, p_z, p_w)^T = P(d_x, d_y, d_z, 0)^T$ 。然后我们必须区分: 如果 $p_w = 0$, 那么我们有后透视平行光 (情形 a)。因此 C 必须是有观察方向 $(p_x, p_y, 0)$ 的平行视锥体, 它只包含整个单位立方体 $[-1, 1]^3$ 。如果 $p_w \neq 0$, 那么我们有后透视光源 (情形 b)。 C 则是原点为 $(p_x/p_w, p_y/p_w, z_\infty)$ 的视锥体, 它包着单位立方体。最后, 如果 $p_w = 0$, 我们有一个光汇点 (情形 c)。假如这样, 我们生成一个光锥体 (light frustum) 就像为前面的情形生成的一样, 但是我们用 -1 缩放 z 来反转深度。所以, C 本质上是一个矩阵, 它也同和单位立方体一样大的场景的标准阴影贴图一起使用。

如前面所描述的第一个自然实现在最初会提供好的效果。然而, 它也会在某些情形下制造弊端, 如质量整体下降, 或者阴影丢失了。接下来, 我们会描述 PSM 的缺陷以及怎样避免它们。

2. 陷阱 1: 闭合的近平面

只有当观察者的视锥体的近平面 (near plane) 不是太近的话, PSM 才发挥令人满意的作用。通常我们可以说如果近平面的距离是 n , 那么阴影贴图分辨率的 50% 被深度范围为 $[n, 2n]$ 的区域使用, 而另外的 50% 用在深度范围为 $[2n, \infty]$ 的区域。如果人们选择一个 1cm 的近平面, 那么观察距离超过 1m 的物体只能得到阴影贴图分辨率的 2%。

因此要尽可能地把近平面推到远处。但若这是不可能的, 例如与其他游戏角色非常接近时, 最好的解决办法是将视锥体虚拟地延伸。近平面和远平面朝前移动一段距离, 实际上是观察者朝后移动了相同的距离, 因此它们匹配原始锥体的近平面和远平面。新的锥体定义了一个新的后透视投影矩阵 P , 我们用它来构造阴影贴图。由于新的矩阵 P 覆盖了一个更大的锥体, 因此它浪费了一些分辨率, 但由于它放大了近平面的距离, 因此它表现得更好了。

3. 陷阱 2: 深光源

现在为止, 我们仍然假定阴影贴图覆盖了整个观察者的视锥体 (在后透视空间的单位立方体)。然而一般而言, 这已经过头了, 因为视锥体从场景中突出, 因此有很大的区域是空的。如果我们把自己限定在视锥体和场景外壳的交叉点, 那么我们能够获得分辨率。

效果如图 5.3.4 所示。在左边的图中, 我们见到了世界空间中的视锥体和场景的边界盒 (bounding box)。在中间的图像中, 我们可以看到在后透视空间中的光源锥体, 它是包围着视锥体的一个高光源和一个深光源的锥体。可以看到对于深光源 L2 来说, 锥体 (虚线部分) 有一个很大的张开的角, 因此退化了。如果我们使光源锥体只是环绕视锥体和场景的边界盒的交点, 那么这能被明显地改善。结果, 深光源锥体的张角变小了并减少了退化。对高光源

L1 来说，这种改善不明显。

对这种构造来说，视锥体和边界盒的交叉点必须被计算。这需要代码来描述一个闭合的凸多面体（closed convex polyhedron）并且来切断一半的空间。我们以场景边界盒开始，随后切断 6 次半空间来定义视锥体。

4. 陷阱 3：视锥体外的阴影投射者

另一个遗留的问题是，阴影贴图必须额外包含所有能够投射一个阴影到视锥体的点。例如，如果一个观察者站在一棵树下，他可能看不到这棵树，但是这棵树肯定能够投射一个可见的阴影，这样它就必须被包含在阴影贴图中。运用目前的算法，这棵树将被光锥体（light frustum）的近平面省略掉一部分。

通过朝光源移动光锥体的近平面，我们能够解决这个问题，这样所有潜在的阴影投射者（caster）就能被包含进阴影贴图。用先前所描述的交叉点（intersection）计算很容易解决这个问题：我们只能切除那些点远离光源的半空间，而不是切除如先前所描述的所有视锥体的半空间；换言之，我们不能切除锥体和光源之间的区域。

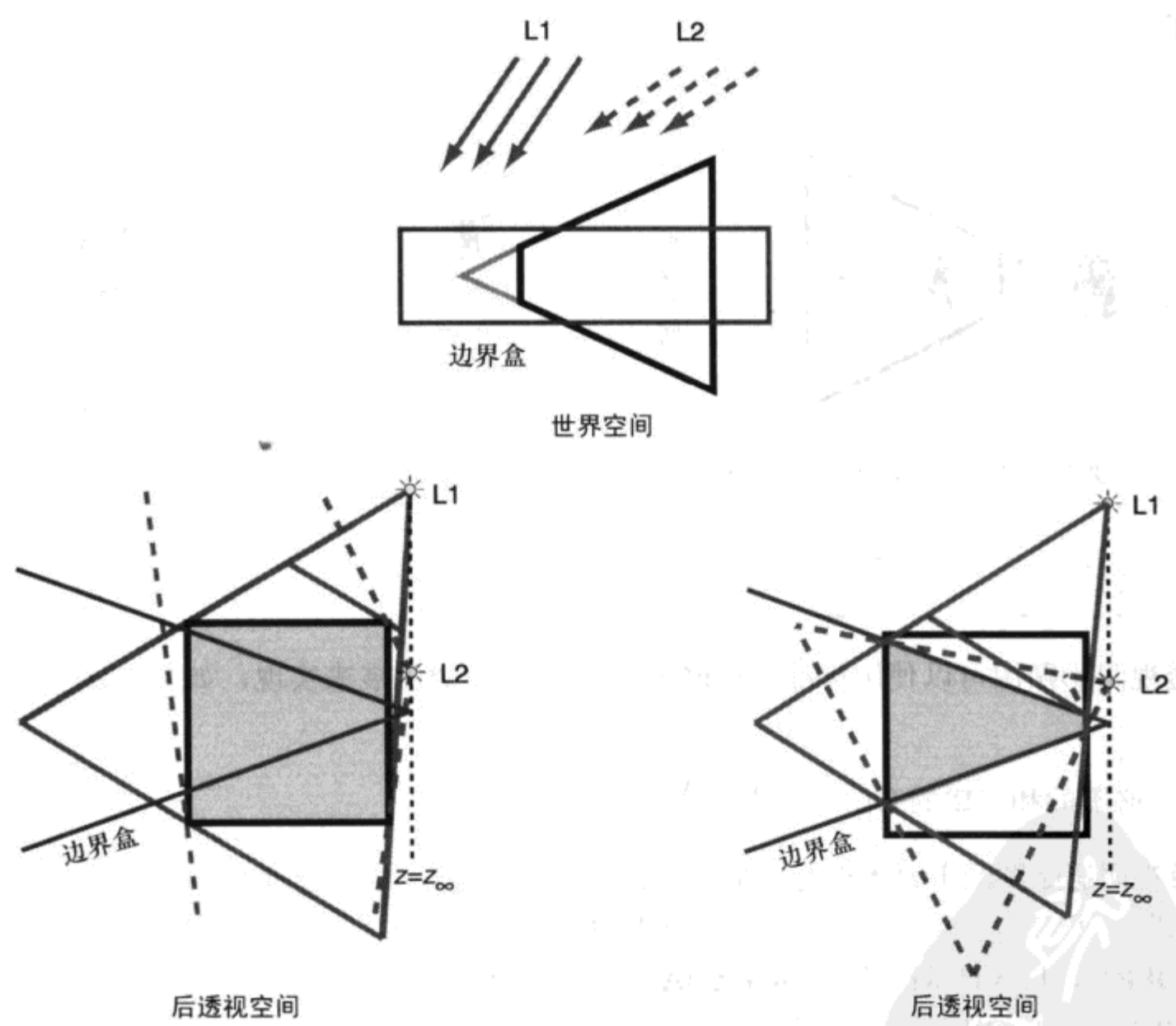


图 5.3.4 世界空间中的一个视锥体和场景的边界盒（左图）。在后透视空间中，因为后透视空间原点非常接近单位立方体，所以一个包含整个视锥体的深光源的光源锥体退化了（中间的图）。如果计算后透视边界盒和视锥体的交点，并且使光源锥体适合这个交点，我们可以获得一个减少退化的深光源的锥体（右图）

最新的图形硬件甚至运用了更好的解决方法。`NV_DEPTH_CLAMP` 扩展避开了关于近平面的剪切，取而代之的是把近平面前面的点投影到近平面上。渲染阴影贴图时用这种扩展，潜在的遮光板不被剪切，因此仍然可见并且能够投射正确的阴影。

5. 陷阱 4：观察者后面的阴影投射者

最后，这里仍然存在一个严重的缺陷，它来源于透视投影的一个不直观的属性。所有在摄像机后面的点被映射到越过无限平面（infinity plane）的点，如图 5.3.5 所示。顺着世界空间中和透视空间中的光线，点的次序在变化，因为光线的无限远处的点被映射到了一个限定的位置。由于这个原因，越过无限平面的点的包含物是难以使用的，例如，需要一个附加的阴影贴图。我们对这个问题的解决方案同解决缺陷 1 中近平面的方法是一样的：我们实际上朝后移动摄像机；换言之，我们朝后延伸视锥体，直到所有需要在阴影贴图中显示出来的场景点在观察者的前面为止。最后，我们从摄像机朝光投射一个光线，并使用场景边界盒计算交叉点。这提供给我们在摄像机后面离摄像机最远的潜在遮光点（occluding point）。由于我们实际上朝后延伸视锥体，因此临界点正好在摄像机平面上。然后我们为这个扩展的视锥体生成了一个透视阴影贴图，因此我们可以确信所有必需点都被包括了。

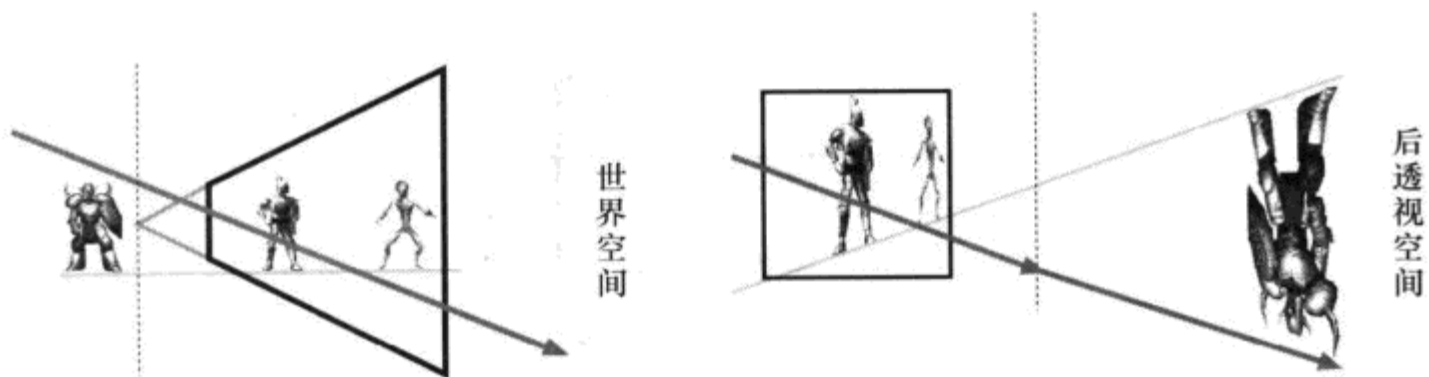


图 5.3.5 摄像机后的物体在后透视空间中被映射到超过了无限的平面（右图），因此造成了沿着光线的点的次序的改变

5.3.5 实现

透视阴影贴图可以使用标准 OpenGL 函数以及少数扩展来实现，如同在这节中所述的一样。

1. 必需的和渴望的 OpenGL 扩展

透视阴影贴图的主要优点之一是，它们可以通过简单地选择另外的矩阵 S 来代替标准的阴影贴图。因此，它们可以在每个支持标准阴影贴图的平台使用；更确切地说，那些支持 `ARB_DEPTH_TEXTURE` 以及 `ARB_SHADOW` 扩展的，都是 OpenGL 1.4 的一部分。

`ARB_SHADOW` 扩展只提供了一个阴影值 0 或者 1，其结果是完全黑阴影。`ARB_SHADOW_AMBIENT` 扩展允许你选择一个非零的阴影结果值，因此阴影只是暗的并不是完全黑，不应该使用。取而代之的是，程序片断（或者寄存器组合）应该被用于定义阴影区域：完全丢弃镜面反射，通过一常数因子定义漫反射，并且保持环境反射不变。只有当三

个反射组件分别提供给程序片断时才有可能, 这样依次需要一个恰如其分的顶点编程。如果使用了顶点编程, 那么为访问阴影贴图的纹理坐标的生成也可以通过顶点编程来计算, 不需要使用难用的 `glTexGen()`。

实验显示纹理贴图应该拥有原始图像两倍的分辨率, 这样只可能使用 `p-buffer` (`GLX_SGIX_PBUFFER` 或者 `WGL_ARB_PBUFFER` 扩展)。此外, 必须注明 `PSM` 是视图相关的, 因此它们需要为每一帧重新生成。这就是为什么优化纹理拷贝运算是非常重要的。大多数阴影贴图的实现使用 `glCopyTexSubimage()` 来拷贝 `p-buffer` 深度贴图到一个阴影贴图纹理。一个更好的可完全避免昂贵的拷贝运算的方法是使用 `WGL_ARB_RENDER_TEXTURE` 和 `WGL_NV_RENDER_DEPTH_TEXTURE`, 它允许我们绑定 `p-buffer` 的深度缓冲到一个纹理, 因此拷贝可以被完全避免。

`NV_DEPTH_CLAMP` 扩展避免了视锥体外遮光板 (occluder) 的丢失, 并使得相关区域 `V` 的计算明显更简单。

2. 伪代码例子

一个典型的透视阴影贴图的实现如下。

```
// 避免太接近近平面的偏移量
nearOffset = max(0, zNearMin - zNear)

// 观察者后面的光偏移量
backlightOffset = calcBackOffset();

// 产生矩阵 P
P = frustumWithOffset(max(nearOffset, backLightOffset))

// 计算相关区域 V
V = intersection(view frustum, scene bounding box)

// 计算后透视光源锥体 C
C = lightFrustumThatSees(postPerspective(V))

// 渲染阴影贴图
setRenderMatrix(C*P)
setRenderTargetShadowMap()
renderSceneWithDepthClamp()

// 渲染视角
setTextureCoordGeneration(C*P)
turnOnShadowMapping()
renderScene()
```

5.3.6 结论

在靠近摄像机的区域通过为阴影采用更高的分辨率, 透视阴影贴图可以提供比标准阴影贴图算法更高质量的阴影。因为在后透视空间运算, 所以这种技术介绍了一些在视锥体外的光和阴影投射者的处理上的微妙技巧。在这里已勾画的实现实例说明了要维持高性能时的大部分问题。

5.3.7 参考文献

[Reeves87] Reeves, W.T., D.H. Salesin, and R.L. Cook, "Rendering Antialiased Shadows with Depth Maps," *Computer Graphics* (Proc. SIGGRAPH 87), 1987, pp. 283–291.

[Stamminger02] Stamminger, M., and G. Drettakis, "Perspective Shadow Maps," *ACM Transactions on Graphics* 21(3) (Proc. SIGGRAPH 2002), 2002, pp. 557–562.

[Williams78] Williams, L., "Casting Curved Shadows on Curved Surfaces," *Computer Graphics* (Proc. SIGGRAPH 78), 1978, pp. 270–274.



5.4 结合使用深度和基于 ID 的阴影缓冲

作者: Kurt Pelzer, Piranha Bytes

E-mail: kurt.pelzer@gmx.net

译者: 刘永静

审校: 谷超

阴影在我们的环境视觉中是很一种很重要的元素。它们的定位和方向提供了有关物体放置和光源位置的详细信息。把阴影加入一个场景中有助于增强真实感并且提供额外的深度提示。

对于实时应用程序来说, 在这方面首先要做的是对用于编码阴影 (encode shadows) 的静态光源贴图 (light map) 进行预先计算。然而, 为了正确地处理动态物体和光源, 必须进行动态阴影计算。对于动态阴影, 目前有几种著名的实时处理技术, 包括投射阴影纹理 (projected shadow texture), 模版阴影体 (stenciled shadow volume) 和阴影映射 (shadow mapping)。每种方法都有其优点和缺点。

这篇文章描述了一种混合的方法, 结合使用了深度和基于 ID 的阴影缓冲这两种技术, 充分利用了这两种方法的优点。这种方法允许自遮蔽 (self-shadowing) 并且支持最高 42 875 个惟一的 ID。

5.4.1 已有的阴影映射技术

这里有多种实时计算动态阴影的方法。其中一种最著名的方法是阴影映射, 它需要两次渲染 (two render passes) 每个阴影投射光源来检测像素是否在阴影中。

1. 第一阶段: 计算缓冲区的值

计算阴影需要计算从光源来看的可见性。从光的视点来看, 所有落到光截面中的对象都被渲染到一个纹理。贴图的每个像素存储了离光源最近的像素深度值或对象的 ID 信息。

2. 第二阶段: 检测阴影

从观察者的角度来说, 帧缓冲中的所有场景对象都要被渲染。把每个像素的深度和 ID 信息与记录在阴影贴图 (shadow map) 中的值进行比较。为了能够进行这个比较, 阴影贴图必须从光投射到场景上。我们需要检测能否从光的位置看到这个片断。无论光是否是照亮的, 在阴影中的对象总

是看不到的。因此，如果测试成功，则这个像素是亮的，反之，它就是暗的。5.4.1 图解说明了这两种情况。

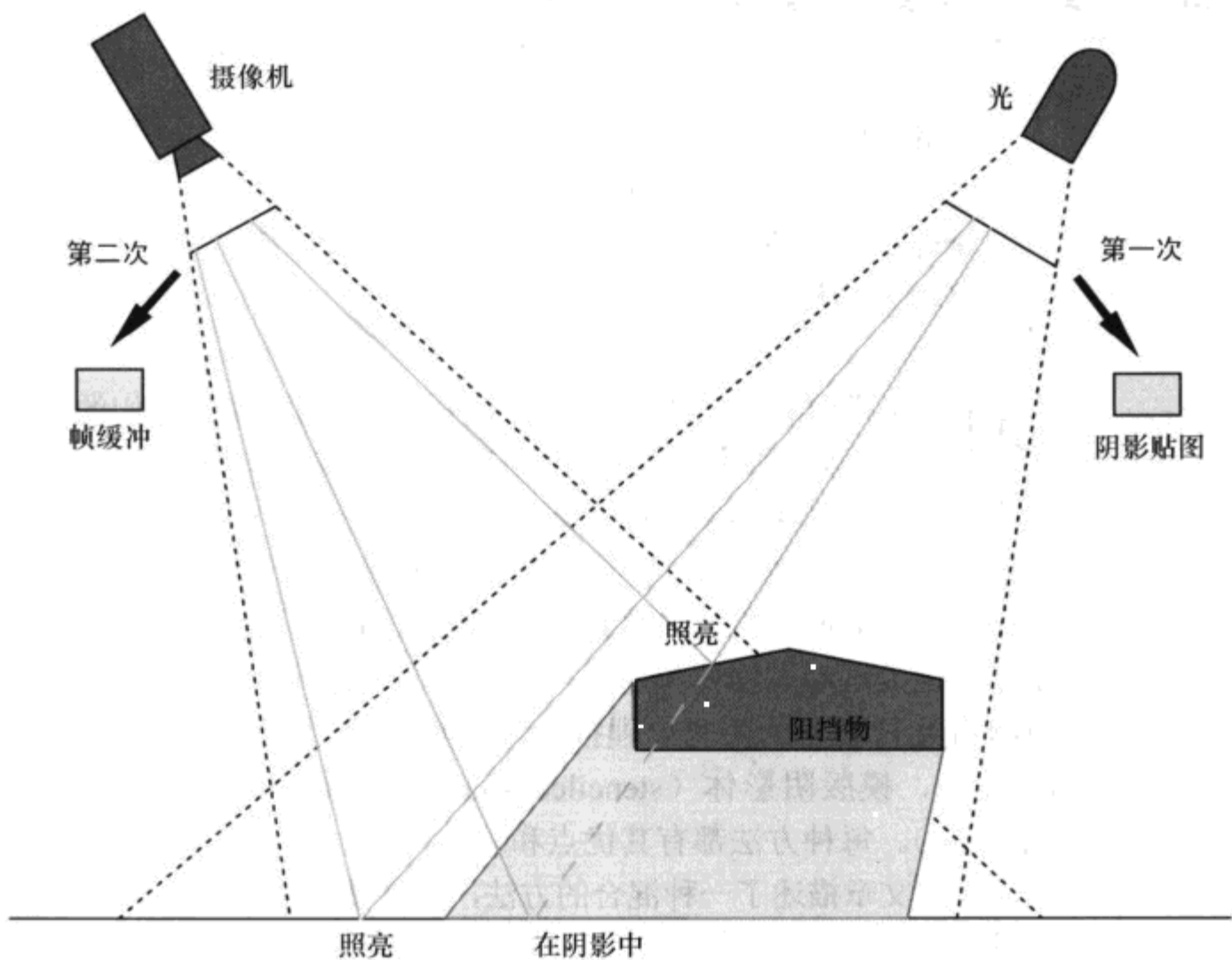


图 5.4.1 普通的阴影贴图技术

5.4.2 深度和基于 ID 的阴影缓冲

让我们来仔细看看最流行的阴影映射 (shadow mapping) 技术：深度缓冲 (depth buffer) [Williams78], 和 ID 缓冲 (ID buffer) ([Hourcade85] and [Dietrich01])。这两种方法非常相似。

对于深度缓冲技术来说，阴影贴图的每个像素包含着距离光最近的深度值 (depth value)。在阴影检测中，这些深度值被用来与那些从观察者的 camera (变换成光照 camera 空间) 所计算的值进行比较。如果 camera 深度值比阴影贴图深度值大，则像素在阴影中，反之，像素不在阴影中。这使对象内部的自我遮蔽 (self-shadowing) 成为可能。有一个缺点是阴影的质量依赖于阴影贴图中深度值的数值精度。当普通用途的硬件 (等于或低于 DX8 的级别) 能够执行这种方法的时候，z-buffer 必须被存储为纹理的一种颜色或者 alpha 通道。但它只提供了 8 位精度，对于通常情况下这样的精度还不足以达到要求。DX9 级别的硬件以及更新的支持渲染 32 位浮点数的纹理。然而，即使是 24 位或者 32 位经常也不足以达到要求的精度来区分两个物体。

ID 缓冲通过精确的对象区分克服了这个精度问题。我们赋予每一个对象一个自己的 ID。这些 ID 必须以物体距离光的远近的排序形式为基础。不是比较 z-depths，而是比较 ID。如果现在的 ID 比阴影贴图 ID 大，则像素在阴影中，反之，像素是亮的。但是，它也有一些

缺点。

- ID 受 8 位精度限制，所以只支持 256 种不同的对象。
- 对象被定义为一种不能遮蔽自己的东西。所以，对象的个别部分不能互相遮蔽，我们失去了自我遮蔽。
- ID 必须基于严格的排序规则，这就是为什么这种技术有时被称为优先缓冲（priority buffer）。我们必须为投射阴影的每一个光进行一次对象排序。

5.4.3 结合深度和 ID 缓冲

基于 ID 的阴影缓冲比基于深度的阴影缓冲能够提供更多有用的优点，反之亦然。这一节略述每一个优点以及我们打算如何运用它们。在这一节中，我们介绍组合的阴影缓冲（combined shadow buffer）。

1. 修改 ID 缓冲的技术

标准 ID 缓冲方法的一个问题是，对象必须根据距离光的远近正确地排序。我们需要这样的排序，因为当对象的 ID 大于 blocker 的 ID 时，对象将只接收阴影。因此，为了能用新的组合缓冲在没有这种排序时能够完成我们的工作，我们必须为 8 位颜色通道开发一个相等的测试。这意味着我们可以简单地通过失败的相等测试（计算得到的 ID 不等于投射的 ID）来检测阴影中的对象（从光的位置看不到的对象）；对象排序不再需要。

标准 ID 缓冲方法的另外一个问题是 ID 数量的限制。组合缓冲的解决办法是在同一时间使用所有的三个颜色通道。此外，我们必须扩展相等测试覆盖所有的三个通道。首先，在每个颜色通道中，测试必须确定一个结果，随后，这些结果必须在 alpha 通道中汇总。最后，alpha 测试能够检测像素是否在阴影中。稍后我们将近距离看看详细的处理步骤。我们将会见到，每通道只能区别 35 个 ID。然而，把所有的三个颜色通道组合起来，结果会产生 $35 \times 35 \times 35 = 42\,875$ 个可能的 ID，这足够用静态 ID 支持所有的对象，甚至是在大场景中。

2. 增加一个内部对象深度测试

现在，我们想要在 alpha 通道中把附加的内部对象深度测试（计算得到的深度是否大于投影的深度）同修改过的 ID 缓冲技术（运行在颜色通道中）组合起来。这允许我们为内部对象深度使用 8 位（这意味着我们能够区别 256 个内部对象层）——同样的精度，当运用标准的深度缓冲技术的时候覆盖整个光的范围。它的结果将会被同用相等的测试得到的结果进行组合，并且在 alpha 通道中汇总。再一次，最后的 alpha 测试检测像素是否在阴影中。

3. 比较

和标准的阴影映射方法比较起来组合的缓冲（combined buffer）技术有一些优点，包括较高的精度（ID 和内部物体深度），占用非常低的处理器资源（不再需要对象排序），而且支持自我遮蔽。它们总结在表 5.4.1 中。

表 5.4.1 阴影映射的比较技术

	ID 缓冲	深度缓冲	组合的缓冲
精度	低 256 ID (标准版本)	低 8 位深度 覆盖光的范围 (标准版本)	高 42 875 ID (简化版本) 8 位内部对象深度 (完整版本)
自我遮蔽	部分 只有表面	是 低精度	是 完整版本
对象排序	需要 低 CPU 占用	不需要 非常低的 CPU 占用	不需要 非常低的 CPU 占用
渲染到纹理	需要	需要	需要

5.4.4 组合的阴影缓冲概述

把所有的这些修改加以考虑，让我们看看组合的缓冲阴影如何工作。

1. 缓冲创建阶段

```
For 每个光
    设置组合阴影缓冲纹理为渲染的目标
    把组合的阴影缓冲纹理清为 0×00000000
    For 在光的视线截面中的每个对象
        设置对象 ID 作为一个颜色（RGB）常量，并且设置对象的内部深度作为 alpha
        使用颜色常量，渲染对象到纹理
```

2. 阴影测试阶段

```
For 每个光
    创建纹理矩阵，以便把顶点从视空间（view space）移动到光可视空间（light's view space）
    For 玩家视截面中的每个对象
        设置物体 ID 为一个颜色（RGB）常量并设置物体的内部深度为 alpha
        选择组合的阴影缓冲为一个纹理
        For 每个顶点
            为投射的组合阴影缓冲计算纹理坐标
        For 对象的每个像素
            把常量 ID 与最近的投射 ID 作比较
            If 常量 ID 不等于最近的投射 ID
                像素在阴影中
            Else
                比较常量深度与最近的投射深度
                If 常量深度>最近的投射深度
                    像素在阴影中
                Else
                    像素是亮的
```

在下面两节中，我们将会近距离看看两个阶段（缓冲创建和阴影测试）并且开始构建基

于固定功能多纹理方法的一个实现。这允许我们在以前的 GPU（那些不是真正可编程的，没有 pixel shader 支持）上使用这些代码。将这些实现转到第一代（Direct3D ps1.3 以及更早的）可编程的 shader 很容易，因为在这些版本中的 pixel shader 指令是固定功能风格并且直到 ps1.4 以及更新的之前还不足以能够被认为是可编程的。

5.4.5 第一次：从光照的视点渲染

第一个步骤是创建阴影贴图——一个渲染目标纹理，包含从光的视点渲染而来的场景。我们组合的阴影映射技术的简化版本写对象的 ID，这可从在每个像素的 RGB 颜色通道中的光源处见到。完整版本的组合缓冲技术额外将内部对象深度写到 alpha 通道（见图 5.4.2）。

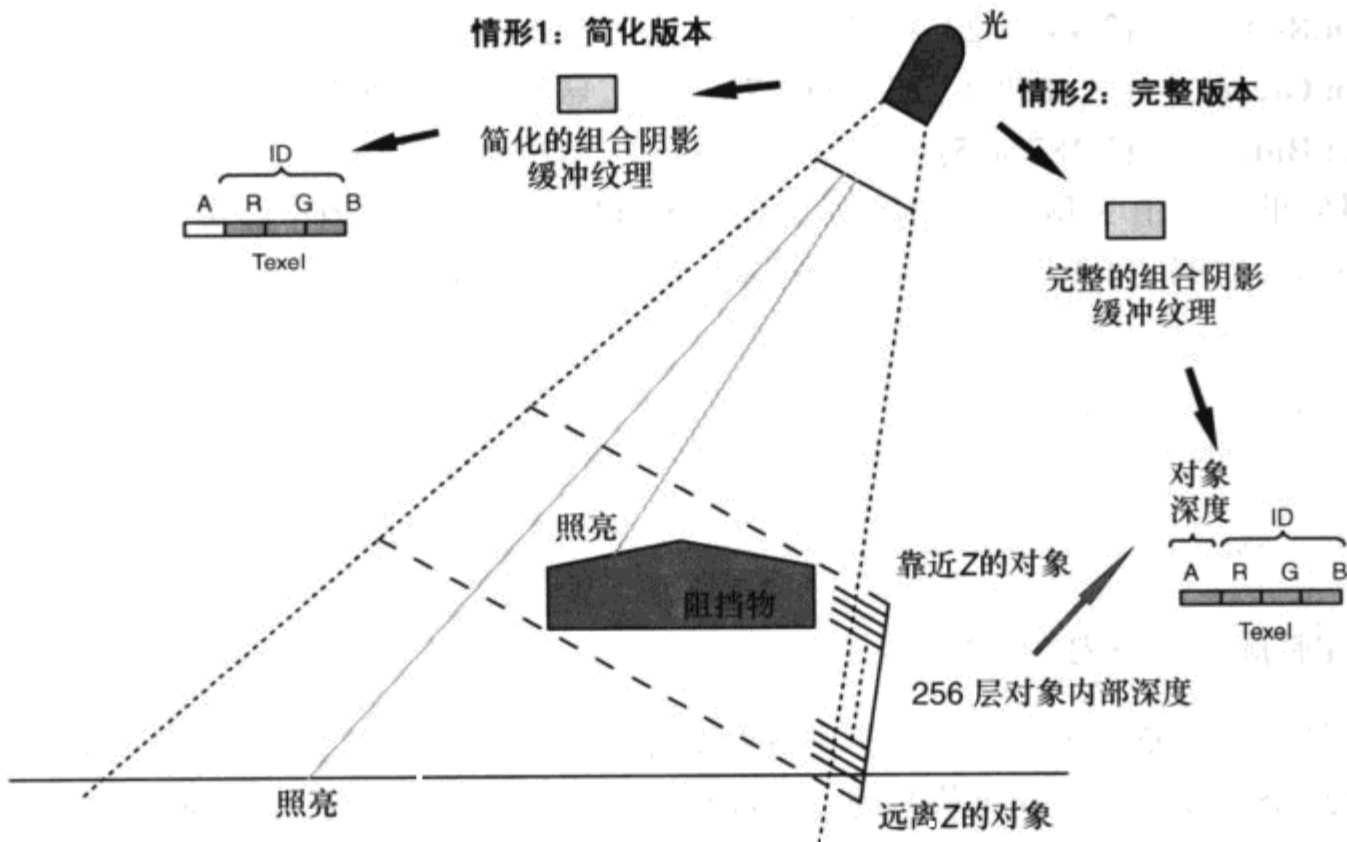


图 5.4.2 来自光的视点的渲染

1. 为什么我们需要为物体 ID 特殊编码

在稍后的阴影检测的阶段，我们将会以点乘积作为一个纹理级的颜色运算来混合已计算的颜色通道在 alpha 通道中的结果。在用来模拟有符号数据之前，这个点乘积把所有的输入（为颜色通道的值）减 0.5 ($y = x - 0.5$)。因此，0x7F 将被解释为 0。然而，因为受这计算的精度限制，0x7F 不是仅有的结果为 0 的数字。在这个主向量（central vector）周围有一个很小的区域也为 0。如果颜色的红、绿、蓝成分都是同在区间[0x79, 0x85]，点乘积结果仍然为 0。临近值 0x78 和 0x86，如果它们在至少一个颜色通道中出现一次，并且都是第一个，会导致非零结果。点乘积是我们的 ID 相等测试过程的一个组件。因此，我们必须考虑它的精度限制并且在至少一个颜色通道中保证一个最小距离为 7 来取得不同 ID。因此，每个颜色通道只区分 35 个 ID 是可能的，总体上所有三个颜色通道有 $35 \times 35 \times 35 = 42\,875$ 个 ID。

2. 简化版（没有为内部阴影提供支持）

就如我们已经见到的，不同的 ID 在它们的至少一个颜色通道中一定有 7 或 7 的倍数的距离（0, 7, 14, 21, 28,...）。因此，用 $0 \leq i \leq 42874$ 编码一个物体 ID 可能看起来如下：

```
// --- 设置对象数据 ---
Color.Alpha = 0x00;
Color.Red   = (i/(35*35))*7;
Color.Green = ((i%(35*35))/35)*7;
Color.Blue  = (i%35)*7;
SetColorFactor( Color );
[ i is in [0,42874] ]
```

例如, $i = 3487$ 被编码为 0x000ECB9A (使用 A8R8G8B8 格式):

```
Color.Red   = (3487/1225)*7           = 2*7   = 14   = 0x0E;
Color.Green = ((3487%1225)/35)*7      = 29*7  = 203  = 0xCB;
Color.Blue  = (3487%35)*7             = 22*7  = 154  = 0x9A;
```

并且它的邻近值 3488 将被编码成 0x000ECBA1（在红色通道中距离为 7）
转换这样的编码 ID 从颜色因子为阴影贴图的像素很容易。

```
// --- Texture Stage 0 ---
TexStage0.SetColorCalc(
    TEXOP_SELECTARG1, TEXARG_COLORFACTOR, TEXARG_CURRENT );
TexStage0.SetAlphaCalc(
    TEXOP_SELECTARG1, TEXARG_COLORFACTOR, TEXARG_CURRENT );
```

3. 完整版（支持内部阴影）

这种技术的完整版本额外的用了内部对象深度。因此，我们不得不在从 0.0 到 1.0 的范围内计算这个深度并且用这个值作为一个纹理坐标来寻址一个简单 depth-to-alpha 映射纹理。

```
// --- 初始化 ---
[ Depth2AlphaTexture: depth-to-alpha 映射纹理
  使用 Width = 256, Height = 1, Format = A8R8G8B8,
  MipMapLevels = 1 ]
Depth2AlphaTexture.LockLayer( &pBits );
Color.Red = Color.Green = Color.Blue = 0x77;
for( unsigned int i = 0; i<256; i++ )
{
    Color.Alpha = i;
    *((DWORD*)( pBits )+i) = Color;
}
Depth2AlphaTexture.UnlockLayer();

// --- 设置对象数据 ---
[ set color factor like shown above in the reduced version ]
matTex.m_Elements.m_f31 = 1.0f/(fObjFarZ-fObjNearZ);
matTex.m_Elements.m_f41 = -fObjNearZ/(fObjFarZ-fObjNearZ);
TexStage0.SetTextureMatrix( matTex );
```

再说，我们必须从颜色因子到阴影贴图的像素转换编码 ID。另外，内部对象的深度必须从深度到 alpha 纹理（depth-to-alpha-texture）中挑选以便把它写到阴影贴图的 alpha 通道中。

```
// --- Texture Stage 0 ---
TexStage0.SetColorCalc(
    TEXOP_SELECTARG1, TEXARG_COLORFACTOR, TEXARG_TEXTURE );
TexStage0.SetAlphaCalc(
    TEXOP_SELECTARG2, TEXARG_COLORFACTOR, TEXARG_TEXTURE );
TexStage0.SetTextureCoordinateCalc(
    TEXCOORDCALC_CAMERASPACEPOSITION );
TexStage0.SetTextureCoordinateTrafo(
    TEXCOORDTRAFO_COUNT2 );
TexStage0.SetTextureFiltering(
    TEXFILTER_POINT, TEXFILTER_POINT, TEXFILTER_POINT );
TexStage0.SetTextureAdressing(
    TEXADDR_CLAMP, TEXADDR_CLAMP, TEXADDR_CLAMP );
TexStage0.SetTexture( Depth2AlphaTexture );
```

5.4.6 第二次：阴影检测

现在我们要对从摄像机视点计算出来的值和阴影贴图中记录的值进行比较。为了能进行这次比较，阴影贴图纹理必须从光投射到场景。因为无论光是什么，只要光不能照到的对象，一定是在阴影中，因此我们想检测是否片段通过计算得到的值与投射值相等（见图 5.4.3）。

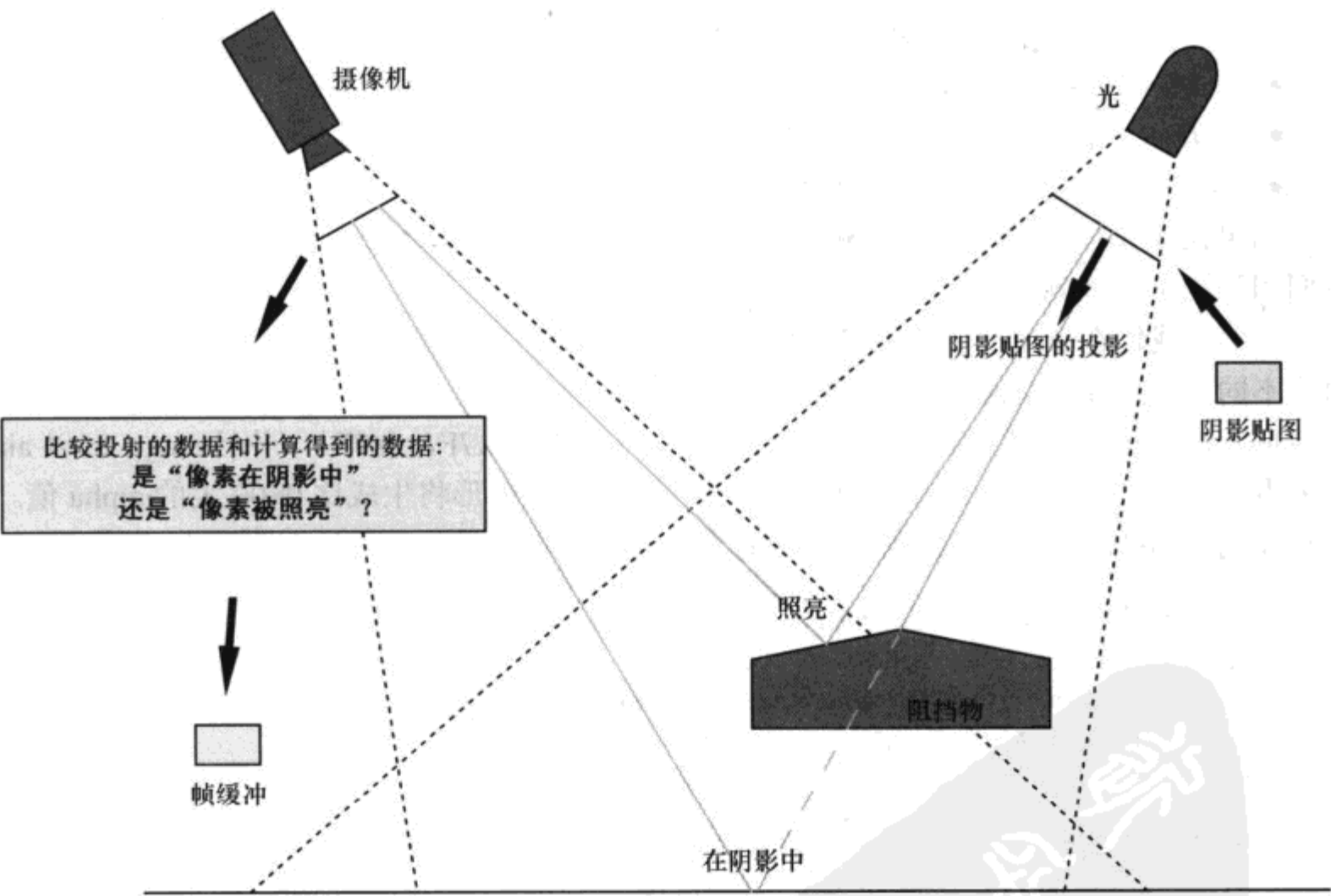


图 5.4.3 阴影检测

接下来的两节呈现了使用特殊设定来为多纹理管道 (multitexturing pipeline) 进行阴影检测测试。以管道输出为基础, 一个简单 alpha 测试将使我们能够检测在阴影中的像素 (例如, 在模版缓冲 (stencil buffer) 中标记它们)。

1. 简化版

就如我们在第一阶段 (从光的视点渲染) 中所见, 很容易以所需要的格式编码对象 ID。然而这一次, 我们必须用加 0x08 来变换在每个颜色通道中的值。

```
// --- 设置对象数据 ---
Color.Alpha = 0x77;
Color.Red   = (i/(35*35))*7 + 0x08;
Color.Green = ((i%(35*35))/35)*7 + 0x08;
Color.Blue  = (i%35)*7 + 0x08;
SetColorFactor( Color );
[ i is in [0,42874] ]
```

增加这个变换值是必须的, 如同能在下面代码清单中多纹理管道 (multitexturing pipeline) 设定中所能见到的一样。没有这个变换值, 在每个颜色通道中, 0 级中的减法只使我们能区分下面两种情况:

(缩写词: $ccIDcc$ = 当前计算得到的 ID 颜色通道; $psIDcc$ = 投射存储的 ID 颜色通道)

- $ccIDcc \leq psIDcc$ ($ccIDcc - psIDcc = 0 \times 00$ 由于夹道)
- $ccIDcc > psIDcc$ ($ccIDcc - psIDcc \geq 0x07$)

通过增加变换向量, 我们能够区别三种情况:

- $ccIDcc = psIDcc$ ($(ccIDcc+0x08) - psIDcc = 0x08$)
- $ccIDcc < psIDcc$ ($(ccIDcc+0x08) - psIDcc \leq 0x01$)
- $ccIDcc > psIDcc$ ($(ccIDcc+0x08) - psIDcc \geq 0x0F$)

如果仅仅让每一颜色通道的减法产生的结果为 0x08, 那么我们有两个同一 ID。这是为我们相等测试的基础。

最后, 我们在 stage1 中把 0x77 加到每一颜色通道中 (通过颜色因子的 alpha 通道) 计算出的不同点, 并且在 stage2 中计算点乘积。对每个颜色通道来说, 只有在 $ccIDcc$ 等于 $psIDcc$ 的情形下, stage 1 才生成颜色向量 $(R,G,B) = (0x7F, 0x7F, 0x7F)$, 并且 stage 2 以 alpha 输出点 $((R,G,B), (R,G,B)) = 0x00$ 为结果。所有其他情形将生成比 0x00 大的 alpha 值。这正是我们的 alpha 测试想要见到的: 0x00 表示亮的像素; 另外所有的在阴影中。

```
// --- 在阴影中的像素将通过这次测试 ---
STRUCT_AlphaTestData alphaTestData;
alphaTestData.m_AlphaTestFunc = ENUM_CMPFUNC_NOTEQUAL;
alphaTestData.m_AlphaTestRef  = 0x00;
SetAlphaTesting( ALPHATEST_ENABLE, alphaTestData );
```

完整的工作流程如图 5.4.4 中所示。这是为多纹理管道设定:

```
// --- Texture Stage 0 ---
```

```

TexStage0.SetColorCalc(
    TEXOP_SUBTRACT, TEXARG_COLORFACTOR, TEXARG_TEXTURE );
TexStage0.SetAlphaCalc(
    TEXOP_SELECTARG1, TEXARG_COLORFACTOR, TEXARG_TEXTURE );
TexStage0.SetTextureCoordinateCalc(
    TEXCOORDCALC_CAMERASPACEPOSITION );
TexStage0.SetTextureCoordinateTrafo(
    TEXCOORDTRAFO_COUNT3|TEXCOORDTRAFO_PROJECTED );
TexStage0.SetTextureFiltering(
    TEXTFILTER_POINT, TEXTFILTER_POINT, TEXTFILTER_POINT );
TexStage0.SetTextureAdressing(
    TEXADDR_CLAMP, TEXADDR_CLAMP, TEXADDR_CLAMP );
[ insert: set the proj. matrix for the shadow map ]
TexStage0.SetTexture( ReducedCombinedBufferTexture );
// --- Texture Stage 1 ---
TexStage1.SetColorCalc(
    TEXOP_ADD, TEXARG_ALPHAREPLICATE|TEXARG_CURRENT,
    TEXARG_CURRENT );
TexStage1.SetAlphaCalc(
    TEXOP_SELECTARG1, TEXARG_CURRENT, TEXARG_CURRENT );
// --- Texture Stage 2 ---
TexStage2.SetColorCalc(
    TEXOP_DOTPRODUCT3, TEXARG_CURRENT, TEXARG_CURRENT );
TexStage2.SetAlphaCalc(
    TEXOP_SELECTARG1, TEXARG_CURRENT, TEXARG_CURRENT );

```

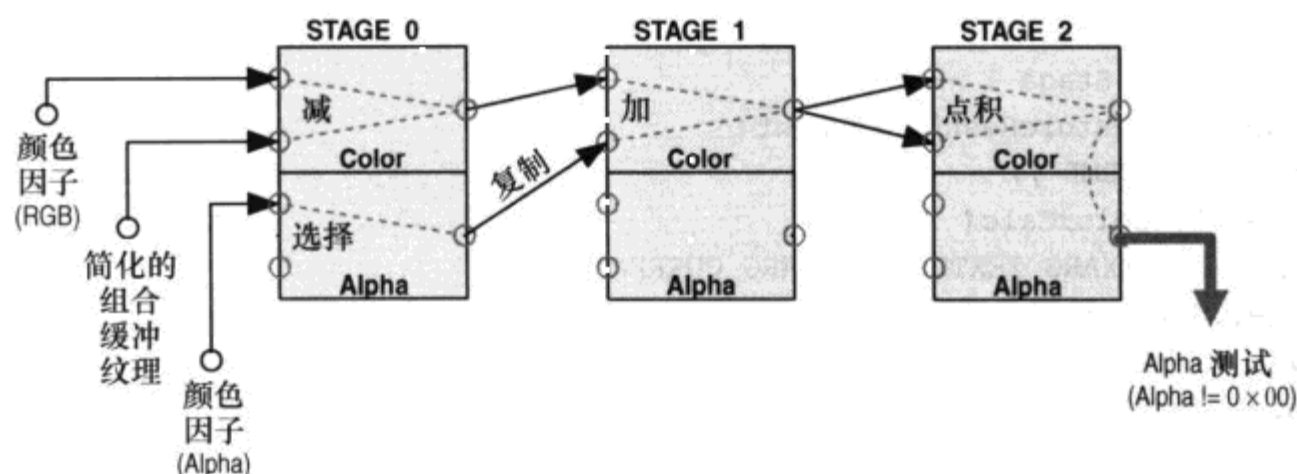


图 5.4.4 简化版的工作流程

2. 完整版

正如在第一阶段中所展示的，内部对象深度也被代入计算。为了写入到阴影图中的深度是可比较的，我们必须从观察者的 camera 空间到光照的 camera 空间变换已计算了的纹理矩阵（为 depth-to-alpha 纹理）。

```

// --- 设置对象数据 ---
Color.Alpha = 0x00;
Color.Red   = (i/(35*35))*7 + 0x08;
Color.Green = ((i%(35*35))/35)*7 + 0x08;
Color.Blue  = (i%35)*7 + 0x08;

```



```

SetColorFactor( Color );
[ i is an element of [0,42874] ]
matTex.m_Elements.m_f31 = 1.0f/(fObjFarZ-fObjNearZ);
matTex.m_Elements.m_f41 = -fObjNearZ/(fObjFarZ-fObjNearZ);
[ matVCSpaceToLCspace: matrix that transforms the
  viewer's camera space to the light's camera space ]
MatrixMultiply( &matTex, &matVCSpaceToLCspace, &matTex );
TexStage1.SetTextureMatrix( matTex );

```

一个在多纹理管道 (multitexturing pipeline) 中阴影检测测试预备的修正版允许我们用在 alpha 通道中的深度测试同 ID 比较的结果组合起来。

```

// --- Texture Stage 0 ---
TexStage0.SetColorCalc(
    TEXOP_SUBTRACT, TEXARG_COLORFACTOR, TEXARG_TEXTURE );
TexStage0.SetAlphaCalc(
    TEXOP_SELECTARG2, TEXARG_COLORFACTOR, TEXARG_TEXTURE );
TexStage0.SetTextureCoordinateCalc(
    TEXCOORDCALC_CAMERASPACEPOSITION );
TexStage0.SetTextureCoordinateTrafo(
    TEXCOORDTRAFO_COUNT3|TEXCOORDTRAFO_PROJECTED );
TexStage0.SetTextureFiltering(
    TEXTFILTER_POINT, TEXTFILTER_POINT, TEXTFILTER_POINT );
TexStage0.SetTextureAdressing(
    TEXADDR_CLAMP, TEXADDR_CLAMP, TEXADDR_CLAMP );
[ insert: set the proj. matrix for the shadow map ]
TexStage0.SetTexture( CompleteCombinedBufferTexture );
// --- Texture Stage 1 ---
TexStage1.SetTextureResultArgument(
    TEXARGUMENT_TEMP );
TexStage1.SetColorCalc(
    TEXOP_ADD, TEXARG_TEXTURE, TEXARG_CURRENT );
TexStage1.SetAlphaCalc(
    TEXOP_SUBTRACT, TEXARG_TEXTURE, TEXARG_CURRENT );
TexStage1.SetTextureCoordinateCalc(
    TEXCOORDCALC_CAMERASPACEPOSITION );
TexStage1.SetTextureCoordinateTrafo(
    TEXCOORDTRAFO_COUNT2 );
TexStage1.SetTextureFiltering(
    TEXTFILTER_POINT, TEXTFILTER_POINT, TEXTFILTER_POINT );
TexStage1.SetTextureAdressing(
    TEXADDR_CLAMP, TEXADDR_CLAMP, TEXADDR_CLAMP );
TexStage1.SetTexture( Depth2AlphaTexture );
// --- Texture Stage 2 ---
TexStage2.SetColorCalc(
    TEXOP_DOTPRODUCT3, TEXARG_TEMP, TEXARG_TEMP );
TexStage2.SetAlphaCalc(
    TEXOP_SELECTARG1, TEXARG_CURRENT, TEXARG_CURRENT );
// --- Texture Stage 3 ---
TexStage3.SetColorCalc(

```

```

TEXOP_SELECTARG1, TEXARG_CURRENT, TEXARG_CURRENT );
TexStage3.SetAlphaCalc(
    TEXOP_MODULATE, TEXARG_COMPLEMENT|TEXARG_CURRENT,
    TEXARG_COMPLEMENT|TEXARG_TEMP );

```

这一次，我们在 stage1 中为每一颜色通道（通过 depth-to-alpha 映射纹理）加上 0x77，并且在 stage2 中计算点乘积。在 stage3 中，我们组合 ID 测试和内部对象深度测试的结果（在 stage1 的 alpha 通道所做）。见图 5.4.5 中的描述。

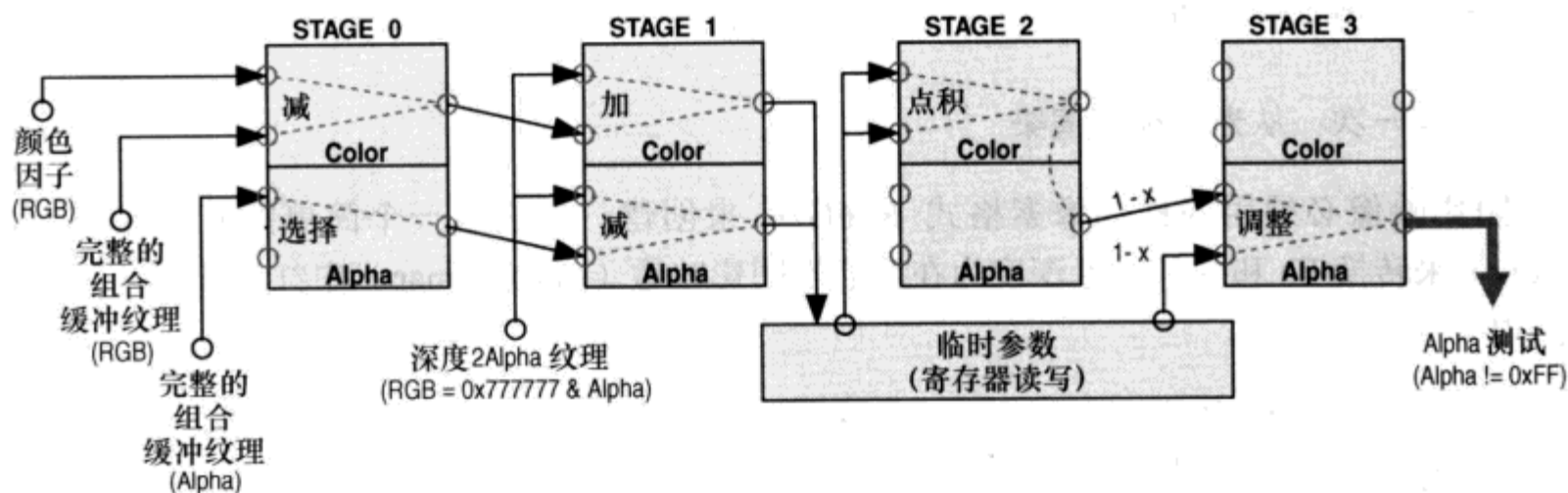


图 5.4.5 完整版的工作流程

为多纹理管道的设定意味着只有在一种情况下，我们将找出在 alpha 输出中的 0xFF: $calculated\ ID = projected\ ID$ 结合了 $calculated\ depth \leq projected\ depth$ 。这正是指示发光像素的情形。所有指示像素在阴影中的其他情形将生成比 0xFF 小的结果。因此，我们的 alpha 测试看起来如下：

```

// --- 阴影中的像素将通过此测试 ---
STRUCT_AlphaTestData alphaTestData;
alphaTestData.m_AlphaTestFunc = ENUM_CMPFUNC_NOTEQUAL;
alphaTestData.m_AlphaTestRef = 0xff;
SetAlphaTesting( ALPHATEST_ENABLE, alphaTestData );

```

5.4.7 在 DX9 2.0 级的阴影中的实现

我们可以在 DX9 级硬件上运用组合阴影缓冲技术来提高精度。在这节中，我们检验一个以 2.0 版像素阴影为基础的实现支持 65 536 惟一 ID 和全 16 位内部对象深度。

我们不必再对对象 ID 进行编码，因为在我们的像素阴影中相等测试要简单的多。用新的像素格式 R16G16，绿色的颜色通道之 16 位精度允许我们支持 65 536 惟一 ID（在我们的特殊 RGB 编码格式中代替 42 875）。而且，这次我们能够支持内部对象全 16 位精度。我们用一个 depth-to-red 映射纹理（mapping texture）（代替 depth-to-alpha）其包含了 2 048 个值（标准 DX9 级硬件支持的最大宽度）来重现区间[0, 65535]。线性纹理过滤（linear texture filtering）计算了一个加权平均值其为在最近的采样点附近。

```
// --- 初始化 ---
[ Depth2RedTexture: depth-to-red mapping texture
  with Width = 2048, Height = 1, Format = R16G16,
  MipMapLevels = 1 ]
Depth2RedTexture.LockLayer( &pBits );
for( unsigned int i = 0; i<2048; i++ )
{
    ColorR16G16.Red = i*32;
    *((DWORD*)( pBits )+i) = ColorR16G16;
}
Depth2RedTexture.UnlockLayer();
```

1. 第一次：从光照视点渲染

阴影映像必须用同样的像素格式 R16G16 来创建。我们用一个简单的着色器 (pixel shader) 来转换 ID 和内部对象深度为在我们的阴影映像 (shadow map) 中红色和绿色的颜色通道中。

```
// --- Pixel Shader ---
// c2 包含在绿色通道中的当前对象 ID
ps_2_0
// 声明用过的资源
dcl    t0          // Depth-to-Red 贴图的 Tex-Coords
dcl_2 s0          // 为 Depth-to-Red 贴图的采样
// 装载纹理
texld r0, t0, s0
// 移动当前的对象 ID 到绿色通道中
mov    r0.g, c2.g
// 设置输出颜色
mov    oC0, r0
```

2. 第二次：阴影检测

在这个渲染阶段，我们用一个着色器 (pixel shader) 来为我们的 ID 运行一个相等测试，和一个为内部对象深度值的更大测试。此外，在固定功能版本中，结果被组合到输出颜色的 alpha 通道中 (0xFF 表示在最终 alpha 测试中亮的像素)。

```
// ---Pixel Shader---
// c2 包含在绿色通道中的当前对象 ID
ps_2_0
// 定义 c0 和 c1
def    c0, 0.0f, 0.0f, 0.0f, 0.0f
def    c1, 1.0f, 1.0f, 1.0f, 1.0f
// 声明用过的资源
dcl    t0          // Depth-to-Red 贴图的 Tex-Coords
dcl    t1          // 阴影贴图的 Tex-Coords
dcl_2 s0          // Depth-to-Red 贴图的采样
dcl_2 s1          // 阴影贴图的采样
// 装载纹理
```

```

texld r0, t0, s0
texld r1, t1, s1
// 计算 Abs(计算得出的 ID - 投影的 ID)
// Max(0, 计算得出的深度 - 投影深度)
mov r0.g, c2.g // r0.r=Calc.Depth & r0.g=Calc.ID
sub r2, r0, r1 // Calculate ID & Depth Differences
abs r1, r2 // r1.g=Abs(Calc.ID-Proj.ID)
max r1.r, c0.r, r2.r // r1.r=Max(0,Calc.Depth-Proj.Depth)
// If( Calc.ID==Proj.ID && Calc.Depth<=Proj.Depth )
// [here: if( r1.g<=0.0f && r1.r<=0.0f )]
// oC0.a = 1.0f
// Else
// oC0.a = 0.0f
cmp r0, -r1, c1, c0 // 运行 ID 和深度测试
mul r0.a, r0.g, r0.r // 把两种测试结果组合起来
mov oC0, r0 // 设置输出颜色

```

5.4.8 结论

这篇文章描述了该如何建立一种处理许多物体 ID 而且支持自我遮蔽的高级阴影映射技术。藉由一个巧妙的纹理阶段状态的组合, 我们可以克服一些旧有的问题并且有效地改进算法的可用性。另外, 这种技术的实现可以在多数图形卡上工作。对于简化版本, 每一个纹理只需要三个纹理阶段, 而对于完整版, 每两个不同纹理我们需要 4 个纹理阶段。



ON THE CD

在随付的 CD-ROM 中有一个组合的阴影缓冲的完整实现。一同包含的是一个用这种技术来计算动态阴影的简单 demo 程序。

然而, 可以有几种途径来改进和扩展我们在这篇文章中所介绍的内容。下面所列出的都是些好主意: [Woo90], [Vlachos00], [Bloom01], [Haines01] or [Haines02], [Stamminger 02] and [Dietrich03]。

5.4.9 参考文献

[Bloom01] Bloom, Charles, and Phil Teschner, "Advanced Techniques in Shadow Mapping," available online at www.cbloom.com/3d/techdocs/shadowmap_advanced.txt, June 2001.

[Dietrich01] Dietrich, D. Sim, "Practical Priority Buffer Shadows," *Game Programming Gems 2*, Charles River Media, 2001.

[Dietrich03] Dietrich, D. Sim, "Robust ObjectID Shadows," *ShaderX2*, Wordware Publishing, 2003.

[Haines01] Haines, Eric, and Tomas Moeller, "Real-Time Shadows," GDC 2001 Proceedings, available online at www.gdconf.com/archives/2001/haines.pdf, March 2001.

[Haines02] Haines, Eric, and Tomas Moeller, *Real-Time Rendering, Second Edition*, A.K.

Peters Ltd., 2002.

[Hourcade85] Hourcade, J.C., and A. Nicolas, "Algorithms for Antialiased Cast Shadows," *Computers and Graphics*, Vol. 9, No. 3, pp. 259–265, 1985.

[Stamminger02] Stamminger, Marc, and George Drettakis, "Perspective Shadow Maps," *Proceedings of ACM SIGGRAPH 2002*, available online at www-sop.inria.fr/reves/publications/data/2002/SD02/PerspectiveShadowMaps.pdf, July 2002.

[Vlachos00] Vlachos, Alex, David Gosselin, and Jason L. Mitchell, "Self-Shadowing Characters," *Game Programming Gems*, Charles River Media, 2000.

[Williams78] Williams, Lance, "Casting Curved Shadows on Curved Surfaces," *Computer Graphics (SIGGRAPH '78 Proceedings)*, pp. 270–274, August 1978.

[Woo90] Woo, Andrew, Pierre Poulin, and Alain Fournier, "A Survey of Shadow Algorithms," *IEEE Computer Graphics and Applications*, Vol. 10, No. 6, pp.13–32, November 1990.



5.5 在场景中投射静态阴影

作者: Alex Vlachos, ATI Research, Inc.

E-mail: Alex@Vlachos.com

译者: 刘永静

审校: 谷超

模版阴影体为活动的角色提供了一个强大的动态阴影解决方案, 可以让这些角色产生固定的阴影。然而, 游戏开发者常常会为动态角色和静态的场景几何选择不同的渲染阴影的方法, 导致这些阴影具有不一致的外观。这篇文章描述了一个直接在静态场景中处理本影的方法, 该方法被用来创建清晰的、固定的场景阴影, 这些阴影具有一致的外观。经过这种方法处理之后, 每个作为结果的多边形都被标记为“在光中 (in light)”或者“在阴影中 (in shadow)”, 以供下一步渲染使用。这样就为场景中的每个细节储存了大量待渲染的完整的阴影体, 产生了具有重要意义的高填充率 (fill-rate) 储存。此外, 我们将简明扼要地讨论一下如何使用低分辨率 (low-resolution) proxy 阴影体在移动的物体上动态地投射阴影。

5.5.1 前期工作

改进过的 Weiler-Atherton 算法 [Weiler77] 也可以提供与本文在这里阐述的方法非常相似的解决方案, 但是如果使用 64 位浮点变量, 由于精度误差的原因, 该算法不能应用于无穷递归。基于该方法的更高级算法已经在《游戏编程精粹 3》的一篇文章中介绍过, 该文名叫“针对复杂数据集计算优化阴影体” [Vlachos02], 不过, 它所描述的方法只能用来计算阴影体的前底面。

本文使用更简单的方法来处理场景中的阴影, 它保留了所有在和不在光中的场景, 而不只是对光可见的多边形。本文提出的这个方法也解决了浮点变量的数值精度问题。

5.5.2 光束基本知识

本文所提出的技术是使用光束来投射阴影。所谓光束就是由空间中的一个点 (发光点) 和一个三角形 (见图 5.5.1) 的三条边所构成的一个金字塔形体积, 它表现为三个平面。

光束, 就像上面所描述的那样, 是指一个完整光束 (full beam)。光束

还有另外两种类型：近光束（near beam）和远光束（far beam）。近光束就是整个光束中在三角形向光面那一边的体积（参见图 5.5.2 左边部分）。而远光束则是指整个光束中在三角形背光面那一边的体积（参见图 5.5.2 右边部分）。这就是说，近光束和远光束是由被发光点照射的三角形来划分的，它们都表现为 4 个面。

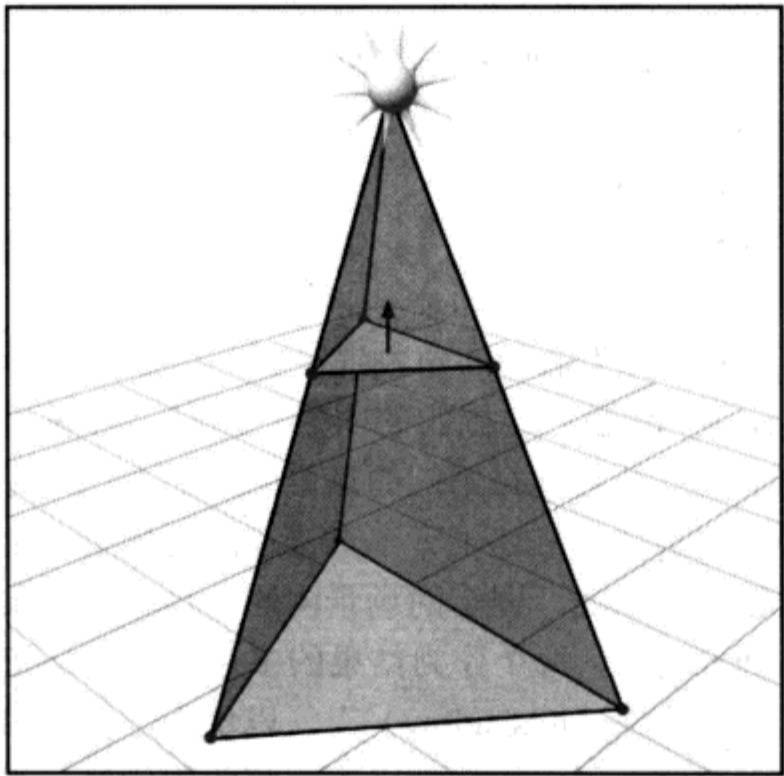


图 5.5.1 从光源和三角形产生的一个完整的光束

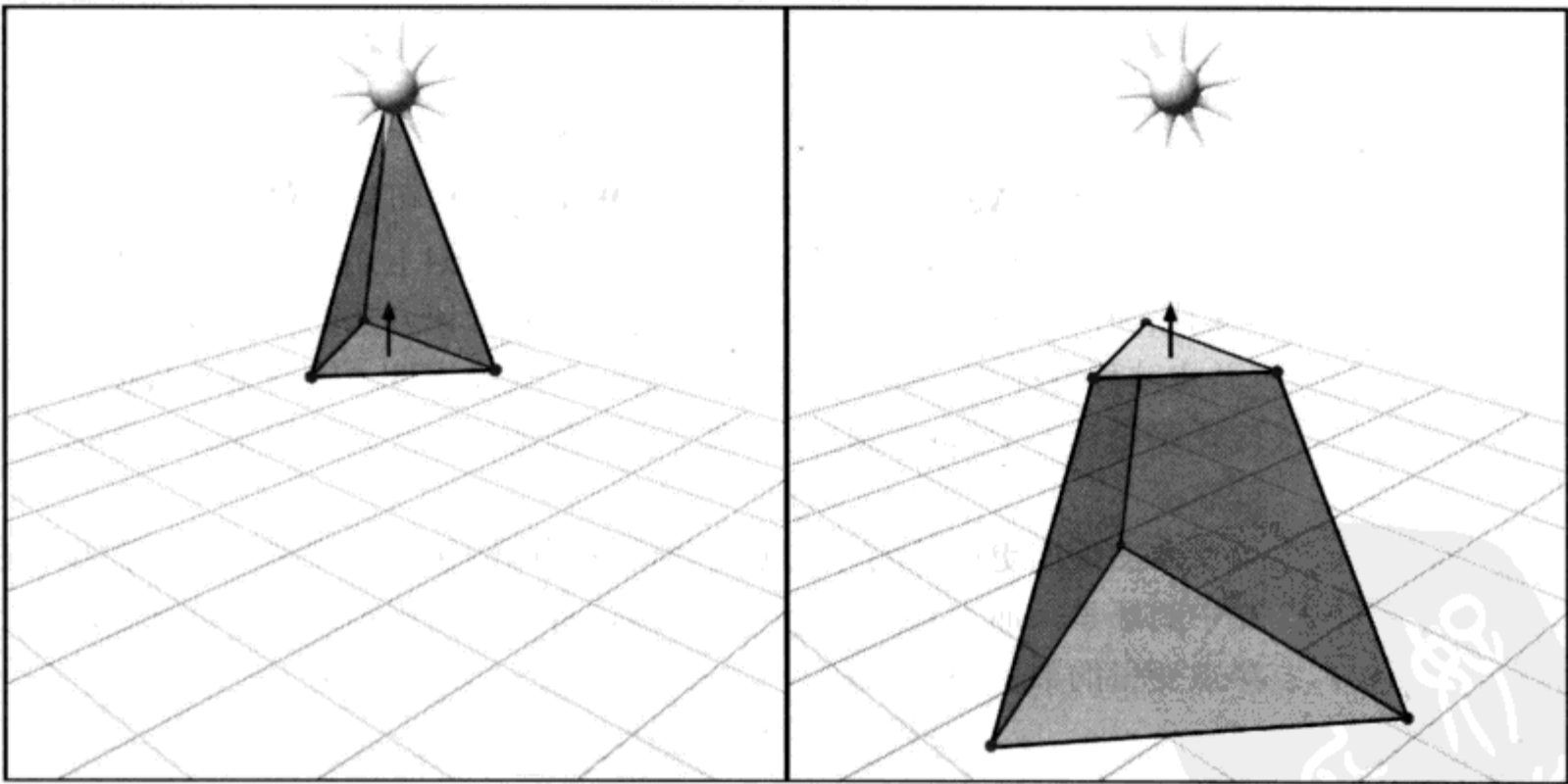


图 5.5.2 （左图）近光束 （右图）远光束

5.5.3 高级算法

我们的最终目的就是用一个新的，切割好的网格模型来代替最初的网格模型。可以使用

好几种方法来达到这个目的，但是考虑到内存存储的因素和简单性，我们使用以下这个最蛮力方法作为我们的首选。

当一个多边形遮挡另一个多边形的时候，实际上就发生了投射。近光源的多边形的远光束将镶嵌到被遮挡的多边形中。构成远光束的 4 个平面都被用于剪裁被遮挡的多边形。当所有的 4 个平面都裁切完被遮挡的多边形的时候，作为结果产生的每个在远光束中的多边形被标上标记，表示其在阴影中。下面的伪代码描述了整个算法。

```
For 场景中的每个阴影投射光源 (L)
{
    For 灯光的平截头体中的每个多边形 (A)
    {
        使用在灯光中的多边形 A 初始化 bin X (多边形数组);
        For 在多边形 A 近光束中的每个原始场景多边形 (B)
        {
            使用多边形 B 的远光束去切割 bin X 中的所有多边形，使用镶嵌后的输出结果去替换 bin X 中的内容;
            If 一个在 bin X 中的多边形同时也在 B 的远光束中，则将它标记为在阴影中;
            优化 bin X 中在光中的多边形;
            优化 bin X 中在阴影中的多边形;
        }
        将 bin X 中的多边形加到输出的多边形数组中;
    }
}
```

沿着原始多边形在灯光平头载体之中部分和灯光平头载体之外部分的边界，解决 T 形连接;

5.5.4 T 形连接

虽然这个阴影剪切算法不会在被镶嵌过的多边形中创建 T 形连接，但是它会在原始多边形的有灯光部分和无灯光部分的分界处创建 T 形连接。例如，想象一下，一个三角形被阴影直接分割成两半，致使它的两个边被分在两部分。如图 5.5.3 中所示，如果三角形的邻近部分对于灯光来说是不可见的，那么该部分的边就不能像它的邻近的面光部分那样被镶嵌。解决这个问题的惟一方法就是在最后手工解决 T 形连接。解决 T 形连接的技术可以在《游戏编程精粹 3》的一篇名为“清除 T 形连接与重新三角化” [Lengyel02] 的文章中找到。

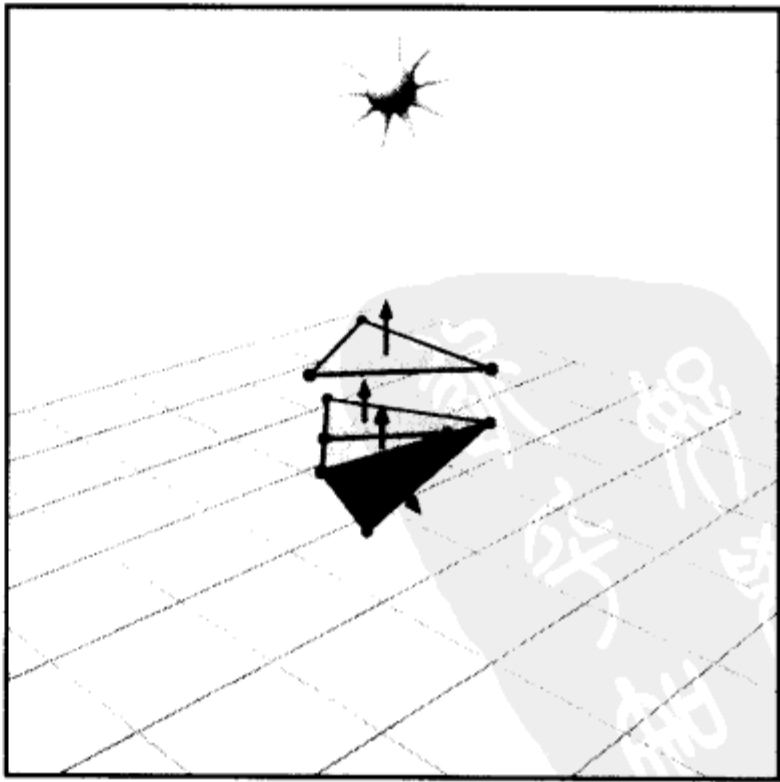


图 5.5.3 沿着面向光源和背向光源的多边形的边界产生的 T 形连接

5.5.5 网格模型最优算法

如果这个算法在没有优化前就被应用到程序中，那么即使在一个相对简单的场景中，应用程序最终也将用光内存。因此，网格模型的优化是必需的。知道在上面的伪代码中哪里可以进行优化是一件非常重要的事情。注意在中间的每个循环中，bin X 被反复地初始化。这意味着在任何时候给出的 bin X 中所有多边形都由同一个原始多边形产生。如果 bin X 中的所有多边形被“粘合”到一起，它们总是可以构成一个来自原始的网格模型的单一多边形。这就允许我们在尝试去优化网格模型的时候，可以做一些关键的假定。

要优化这些网格模型，我们可以进行以下两步操作，顶点移除和边缘塌陷。在开始边缘塌陷这步之前，我们首先应对整个网格模型进行顶点移除。由于我们的多边形网格模型源自单一的三角形，因此我们不必在进行顶点移除和边缘塌陷这两步之前，去操心用于测试的纹理坐标、法线、平面等式等等。这些顶点中的每个顶点数据都是由三角形的三个顶点通过线性组合而得到的，因此，所有的顶点都共享同一个面法线。

由于这些多边形都被标记为“在光中”或者“在阴影中”，因此为了保护我们在这个过程中所创建的阴影边界，它们必须被分离开来优化。

1. 顶点移除

这个优化方法只用于位于原始多边形内部的顶点。

(1) 必须有一张包含网格模型中所有边的信息的表。表的每个入口应该包含该边的两个顶点和共享该边的一个或者两个多边形。

(2) 对每一个惟一的顶点来说，首先应选择一个包含此顶点的三角形作为起点，然后使用边表绕着该顶点顺时针方向“走”到下一个与起始三角形相邻的三角形，此三角形也必须包含此顶点。继续绕着该顶点顺时针方向走，直到到达最初的起始三角形。如果不能够到达起始三角形，那么此顶点就不能应用顶点移除。回到第二步继续进行下一个顶点的判断。如果能够到达起始三角形，则进行第三步操作。

(3) 移除顶点并往左后镶嵌 n 度。一个健壮的镶嵌算法必须避免错误，关于这个方法的例子你可以在[deBerg00]中找到(参见图 5.5.4)。

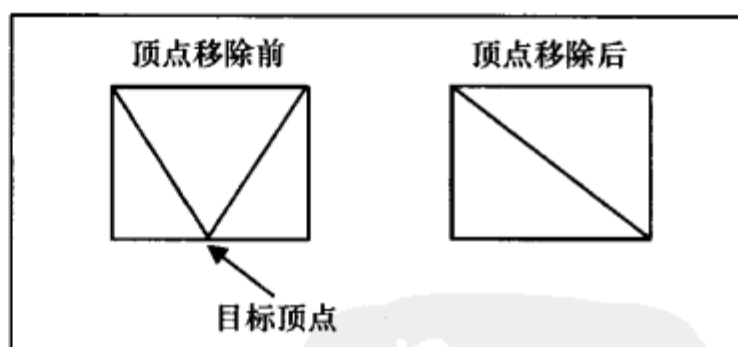


图 5.5.4 顶点移除的结果

2. 边缘塌陷

这个优化方法可用于网格的所有顶点。

(1) 就像在顶点移除的第一步一样，必须有一张包含网格模型中所有边的信息的表。表的每个入口应该包含该边的两个顶点和共享该边的一个或者两个多边形。

(2) 对每一个惟一的顶点来说，首先应选择一个包含此顶点的三角形作为起点，然后使用边表绕着该顶点顺时针方向“走”到下一个与起始三角形相邻的三角形，此三角形也必须

包含此顶点。继续绕着该顶点顺时针方向走，直到到达一个跟第一个边共线的边。如果没有到达这个边或者你又走回到了起始顶点，那么此顶点就不能应用顶点移除。回到第二步继续进行下一个顶点的判断。

(3) 移除顶点并往左后镶嵌 n 度。一个健壮的镶嵌算法必须避免错误，关于这个方法的例子你可以在[deBerg00]中找到（参见图 5.5.5）。

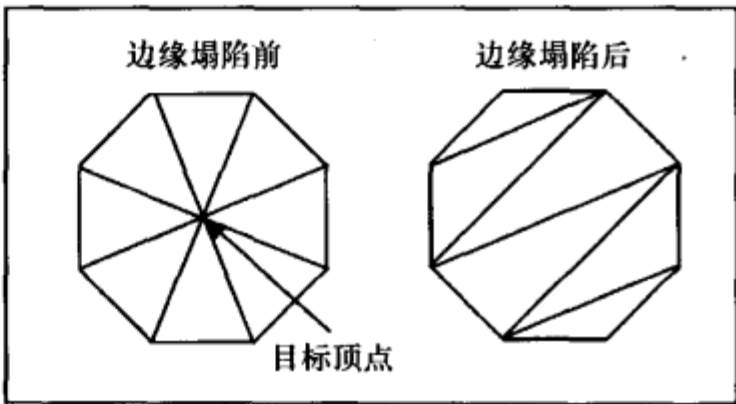


图 5.5.5 边缘塌陷的结果

5.5.6 实现细节

虽然这个算法相当简单，但是实现起来简直就是噩梦。最难的部分就是让优化的代码完美地运行。如果不考虑优化那一步，那么算法剩余部分的实现对有经验的程序员来说，编码不会超过一个下午的时间。对算法的代码进行优化，让它能够在需要达到 64 位浮点精度的条件下成功地运行。

1. 逻辑拓扑与数值精度

开发人员必须接受这样一个事实，那就是对这种类型的算法来说，在 64 位浮点的情况下没有足够的数值精度。严格来讲，使用三角形的三个顶点并不罕见。而且，对顶点来说，它们只是在 64 位浮点的一些极少的位上的不同也是很平常的。在这些问题中，对共线性进行数值测试或者对平面法线的数值计算都是不够的。沿着这些边进行剪裁将只能产生退化的边，不过只要它预先被假定，那就是好的。

2. 边的标志

为了成功地实现边缘塌陷的优化，我们必须使用数值测试代替拓扑测试。但由于数值共线性测试不可行，因此需要转换一下方法。当一个多边形被远光束的平面镶嵌后，所有的边就在这个给出的平面上产生了。这些产生的边中，有一些边在原先平面中如果是属于同一条边的，那么这些边就用同一个 ID 表示。如果不属于原先的平面的（即新产生的边），那么就用它们自己的 ID。在数值共线性测试不可行的情况下，这个思路为检测拓扑的共线性提供了一个方法。

3. 水密网格模型和剪切

依赖网格的拓扑需要连接属性的知识。需要遵循的最简单的规则就是保持网格模型“水密 (watertight)”。简单就意味着在原始的网络模型中没有产生 T 形连接。我们可以从两方面来讲述它的简单性。

维持网格模型水密性的第一个手段就是剪切平面的利用。开发者常常试图去做最优化的事情，并且仅仅为了去剪切那些绝对需要被剪切的多边形而实在地优化他们的剪切算法。这就可能导致 T 形连接。其实，任何永远不会创建 T 形连接的剪切算法都是可接

受的。

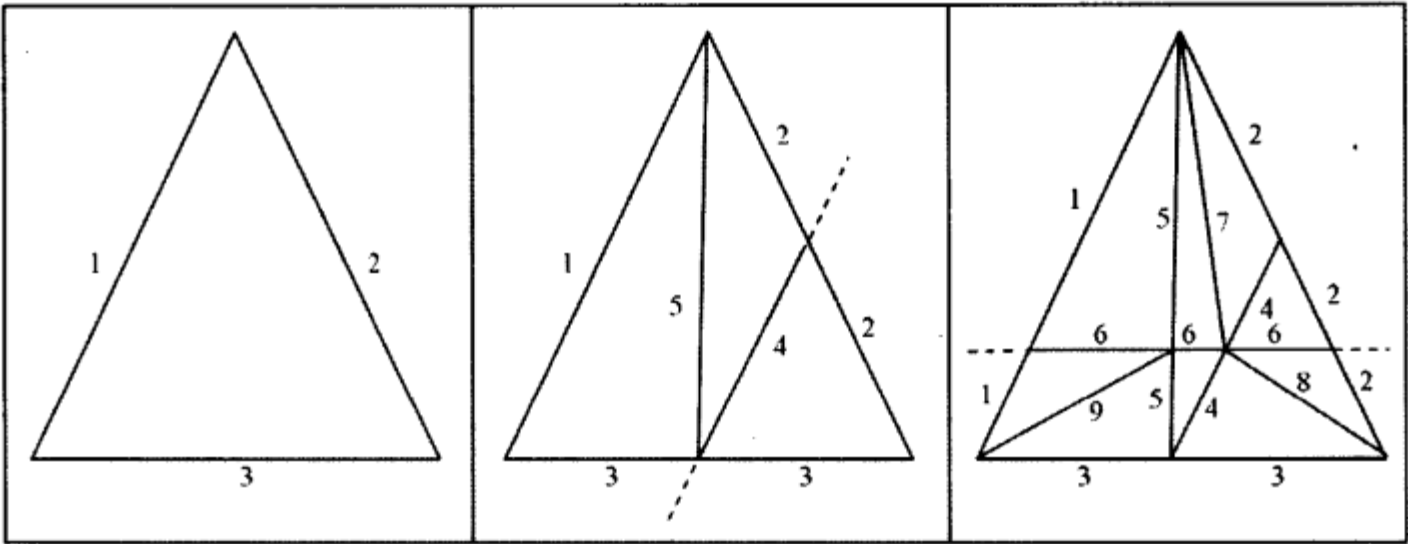


图 5.5.6 图像中所有的数字代表边的 ID。左图为一个单一的三角形。中间为第一次剪切平面后的多边形，使用 ID=4 的线段来切割平面。右图为第二次剪切平面后的多边形，使用 ID=6 的线段来切割平面

由于算法的属性，必然带来维持网格模型水密性的第二个手段，但是它值得提及。当多边形被切割后，沿着阴影边界就会有一些不是顶点移除候选者的顶点产生，因为在光中和在阴影中的多边形是被分开优化的。边缘塌陷首先会在边的有光的那一边产生，创建一个 T 形连接。这不是问题，因为算法中的下一步恰好是优化阴影中的多边形。马上，一个相似的边缘塌陷将在顶点在阴影中的那一边产生。边缘塌陷将沿着一个阴影边界整理自己。

5.5.7 阴影中的动态物体

就像前面提到的那样，这项技术的一个目的就是提供静态场景阴影和动态模版阴影的相容整合性。如果这个离线的阴影切割算法被应用到一个场景中，那么场景中的对象将很明显地不会投射阴影到场景中的动态对象上。在很多案例中，这些或许是不被注意的，但是我们需要有一个在案例中的通用解决方案。

为静态阴影使用这个方法的幕后动机之一是，你可以为影响动态对象的阴影体使用更多的低分辨率场景。在你的场景中，这样做可以非常显著地减少转换负担和透支。此外，阴影体只需要在动态对象在光的平截头体中时被绘制。依靠应用程序，在性能上可以得到巨大的恢复。

5.5.8 结果

图 5.5.7 显示了在一个相对比较密集的环境中应用静态阴影算法的结果。注意在第一层上有多少内部的边和顶点从阴影中被移除。

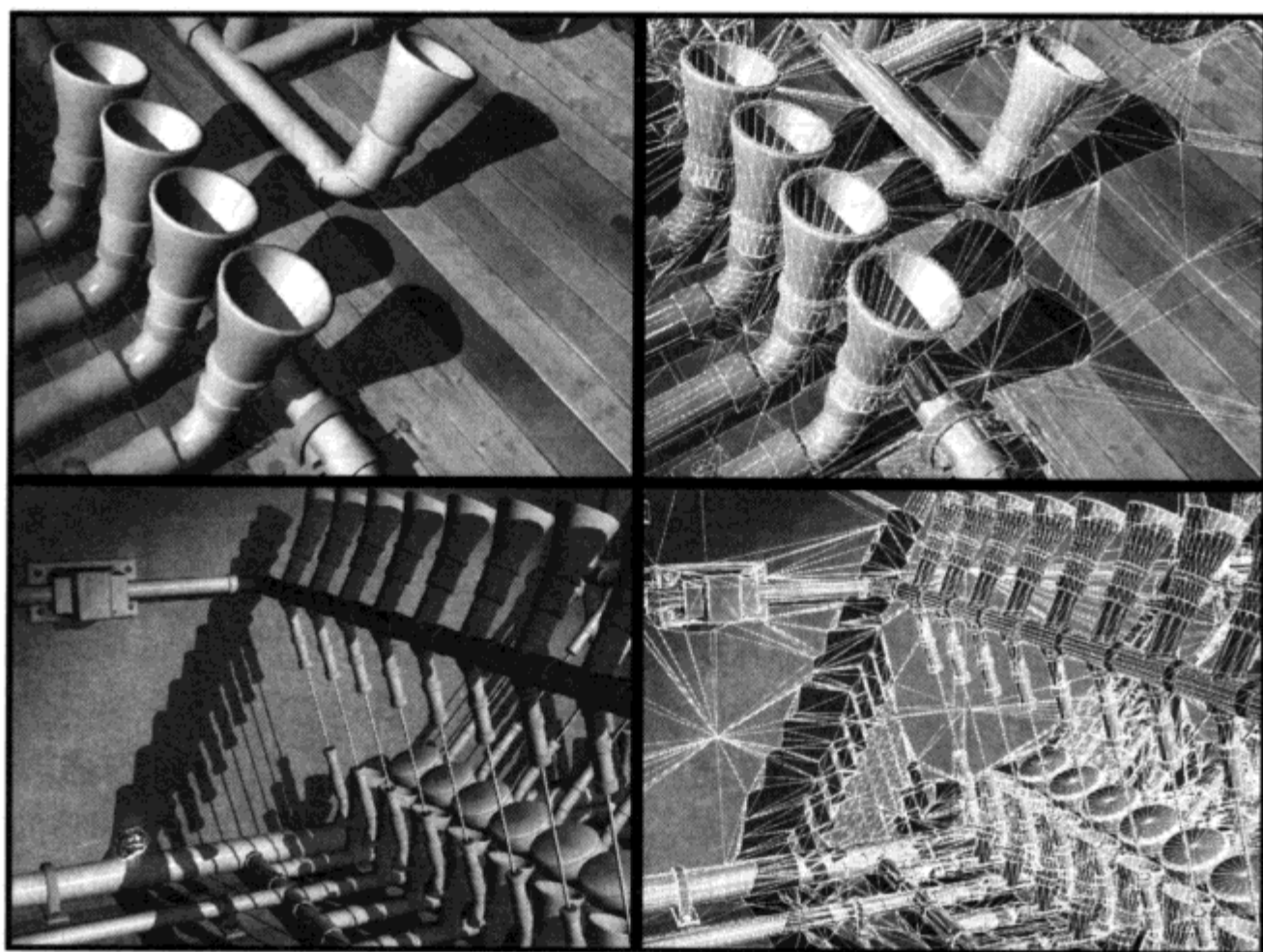


图 5.5.7 ATI RADEON 9700 管道鼓的两个场景。左边是普通屏幕截图，右边是本文结果的线帧显示

5.5.9 结论

本文讨论了在场景模型中直接裁切固体阴影的方法。它为场景提供了统一的外观，并且为动态的角色和对象使用了模版阴影体积。此方法结合了当今游戏中经常使用的模版阴影体，减少了内存透支的瓶颈，最终提高了帧频。

5.5.10 参考文献

[deBerg00] de Berg, Mark, et al., “Polygon Triangulation,” *Computational Geometry Algorithms and Applications, Second Edition*, pp. 45–61, 2000.

[Lengyel02] Lengyel, E., “T-Junction Elimination and Retriangulation,” *Game Programming Gems 3*, Charles River Media, 2002.

[Vlachos02] Vlachos, A., and D. Card, “Computing Optimized Shadow Volumes for Complex Data Sets,” *Game Programming Gems 3*, Charles River Media, 2002.

[Weiler77] Weiler, K., and P. Atherton, “Hidden Surface Removal Using Polygon Area Sorting,” *Computer Graphics*, Vol. 11, pp. 214–222, 1977 (SIGGRAPH ‘77).

Figures provided by Eli Turner, ATI Research, Inc.

5.6 为阴影体和优化的网格模型调整实时光照

作者: Alex Vlachos 和 Chris Oat, ATI Research, Inc.

E-mail: Alex@Vlachos.com, coat@ati.com

译者: 刘永静

审校: 谷超

光照整个网格模型是一件很容易让人理解的事, 被光照后的网格模型会呈现出一个不真实的, 令人意想不到的景象。可是, 当越来越复杂的渲染系统优化那些大量需要被渲染的多边形的时候, 就像背、面选择(从光的观点), 人们提出了照明物这个概念。当完成基于面法线的选择和基于顶点法线的光照后, 就需要做出一些调整。由于阴影体的突出是基于面法线的, 因此照明物具有同能够自身产生阴影的使用模版阴影体积的网格模型相同的应用。此外, 因为法线跟网格模型的几何形状无关, 所以凹凸贴图需要进一步调整。本文探索了这些问题, 并且提出了一个完全可用于像素阴影的解决方案。

5.6.1 光照问题

给出一个被光照射的基于顶点法线的网格模型, 如果要解决光照问题, 那么了解顶点法线和面法线的关系就非常重要。图 5.6.1 显示了两个边界多边形的一条边以及它们的合成顶点法线如何被用来解决光照问题。面法线取 $\vec{N} \cdot \vec{L}$ (矢量表示, 下同) 的值的相反数, 而通过该面的合成顶点法线则生成一个正值。这个差异是一些算法产生问题的原因。

5.6.2 在面法线上操作

与照明有关的某些操作在网格模型上的应用优先于在光栅上的应用。这些操作经常需要一个网格模型可用来进行基于面法线的处理或者修改, 那么稍后就可以基于合成顶点法线进行光照。为了避免产生杂乱的光线, 这个差异需要补偿。

1. 阴影体的突出

面法线用于决定一个网格模型该如何突出其阴影体 [Brennan02]。由于这个原因, 而不是因为网格模型侧面上的柔和光照, 产生了一条像阴影边界的清晰的线条。就像图 5.6.1 中所见到的那样, 多边形 A 最终在阴影

体中，不能接收到光，而多边形 B 可获得很好的光照。这将在两个多边形之间导致杂乱的光照边界。图 5.6.2 在一个使用单一阴影体积进行球体的渲染中阐明了这一点。

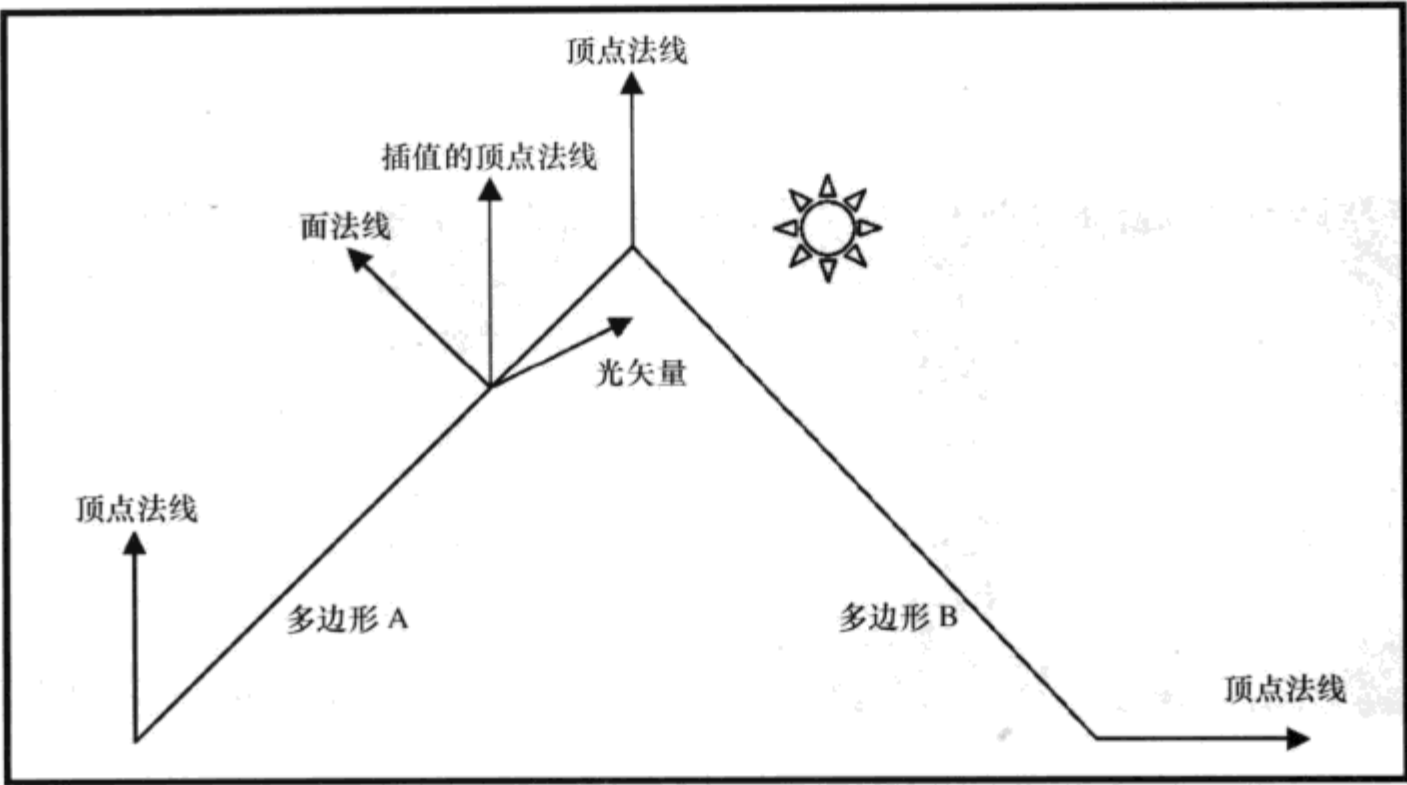


图 5.6.1 如果用面法线来计算通过多边形面的光贡献度，那么将没有光贡献度可供计算。使用合成的顶点法线可以探测贡献到多边形上的光。选择多边形 A 是因为它是背朝光的，这可以导致跨越多边形 A 和 B 边界的光照景象的产生

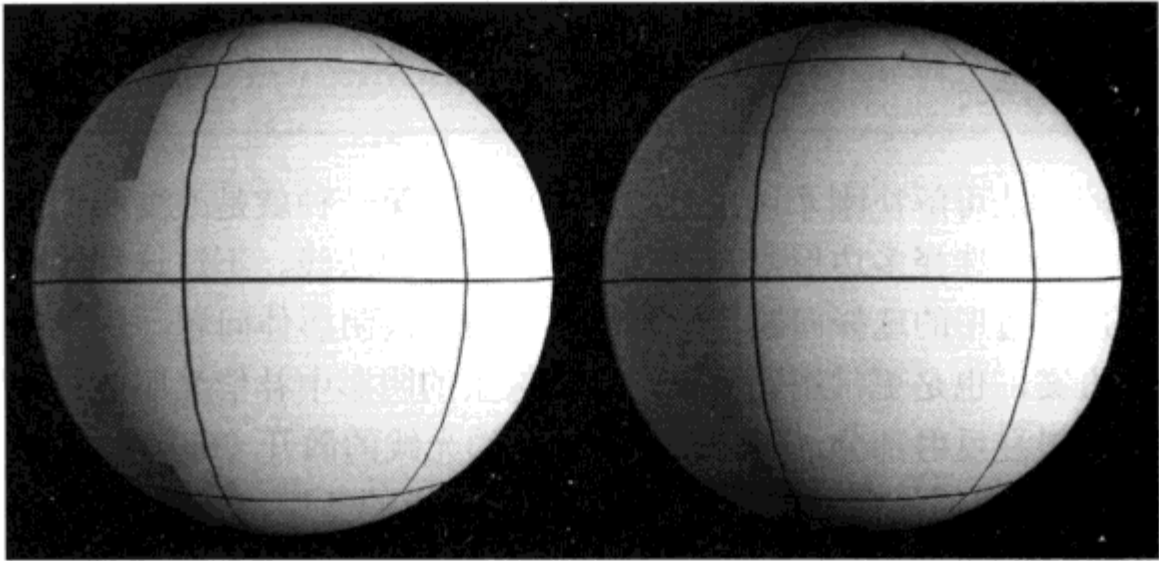


图 5.6.2 （左边）一个被照射的球体，使用顶点光照和阴影体积。顶点光照使用顶点法线来计算照明，而阴影体积则依靠多边形面法线。这导致了在阴影边界产生了光照的不连续。（右边）调整后的顶点光照，因此光线的散开过渡平稳地越过阴影体积边界

2. 多边形的选择

选择那些使用可编程图形加速器的当代图形引擎可能是为了根据影响给定多边形的光线的数量来拣选场景。例如，所有受零度角光线（zero light）影响的多边形通过使用一个假设为零度角光线（zero light）的阴影（shader）被绘制出来，而受单一光线影响的多边形则通过使用一个假设为单一光线的阴影被绘制出来。图 5.6.3 中的最左边的球体是静态场景几何的

一个例子，由于它已经被预处理过了，因此它的多边形已经根据照亮每个多边形的静态场景光线的数量被分组了。因此，在本案例中，多边形将根据它们的面法线和光源的方向被装配起来，因为面法线不必符合用来计算漫射光线的法线，所以将产生光线的不连续性。在图 5.6.3 中间的球体上可见到这种光线不连续性。沿着分开双光阴影（two-light shader）和单光阴影（one-light shader）的边界的一部分，可以看到一条清晰的光线边。

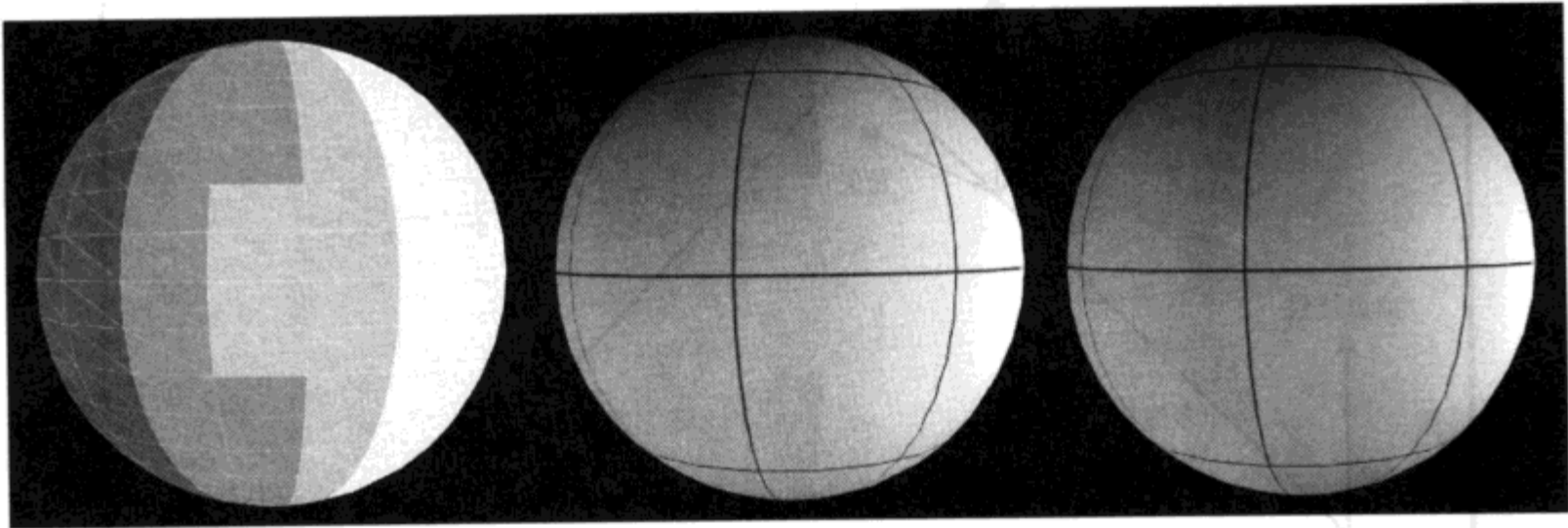


图 5.6.3 （左边）根据照亮多边形的场景光数量，被分成多个网格模型的球体。最亮的阴影有三个起作用的场景光，而最暗的阴影则没有起作用的场景光。（中间）使用与起作用的光源的数量相符的阴影照明每个网格模型。

在预处理期间使用面法线来决定光源的贡献度，而光照的计算则是在运行时（runtime）期间使用导致光照不连续的顶点法线

5.6.3 调整漫射光照

有两种合理的方法可解决刚才讨论过的这些问题。第一种就是改变我们的选择算法为根据所有的三条顶点法线选择多边形，而不是根据单一的面法线。不过这种改变之后的算法虽然可以很好地解决多边形的选择问题，但是它不能够解决阴影体问题。

更好的解决方案，也是更常用的方案，是在像素的阴影中补偿光照的不连续。其基本的思路是测量并且倾斜漫反射部分，从而创建一个新的光线的散开（falloff）角度。如果光线照射的角度是 90°，那么根据经验，我们就会选择更小一点的角度，一般大约为 75°。这种光照调整可以通过基于合成的顶点法线或者法线贴图（normal map）的逐像素法线（per-pixel normal）的光照来完成。在以下两段中我们将来研究一下合成的顶点法线和法线贴图的逐像素法线。

1. 合成的顶点法线

下面的例子是 DirectX 9 HLSL 代码，用来测量和倾斜漫射光照的贡献度。

```
float ComputeDiffuseAdjustment (float diffuseNdotL)
{
    return saturate ((diffuseNdotL * (5.0f/4.0f)) -
                    (1.0f/4.0f));
}
```


上述的函数把 $\vec{N} \cdot \vec{L}$ 作为一个参数（本问题中， \vec{N} 是顶点法线），并且强加一个初始的散开值。从本质上说，在 0.25~1.0 范围内的原始 $\vec{N} \cdot \vec{L}$ 的结果被拉伸以便适合 0~1 的范围。一旦 $\vec{N} \cdot \vec{L}$ 得到了合适地调整，返回值就可以调整用于光线色彩、材料基本色等等。

2. 法线贴图的逐像素法线

因为法线贴图跟它们所在的几何拓扑有很大的不同，所以从法线贴图取样而来的法线处理起来也有些不同。在本例中，我们想要实现一个和为合成的顶点法线实现的算法相似的算法；然而，我们只想计算一个杂乱的散开值（falloff）来掩饰凸起的光照部分。下面的 HLSL 函数可以在使用一个法线贴图的时候调整漫射光照。

```
float ComputeDiffuseBumpAdjustment (float diffuseNdotL,
                                     float diffuseBumpNdotL)
{
    float adjustment =
        ComputeDiffuseAdjustment(diffuseBumpNdotL);
    adjustment *= 1.0f - (pow (1.0f -
        ComputeDiffuseAdjustment(diffuseNdotL), 8.0f));
    return saturate (adjustment);
}
```

从上面的代码中可以看出，此函数需要调用两个参数。它调用的第一个参数的用法跟前一部分（合成的顶点法线部分）中是一样的，除了在这里赋予函数的是使用基于像素贴图的逐像素法线的漫射光照，而不是使用规则的顶点合成的漫射光照。因为法线贴图包含许多不同的法线，所以我们也需要介绍一下基于合成的顶点法线的哈希散开值。通过实验观测，我们发现调整后的漫射光照值的相反数被提升了 8 次幂后所产生的结果是可以接受的，如图 5.6.4 所示。

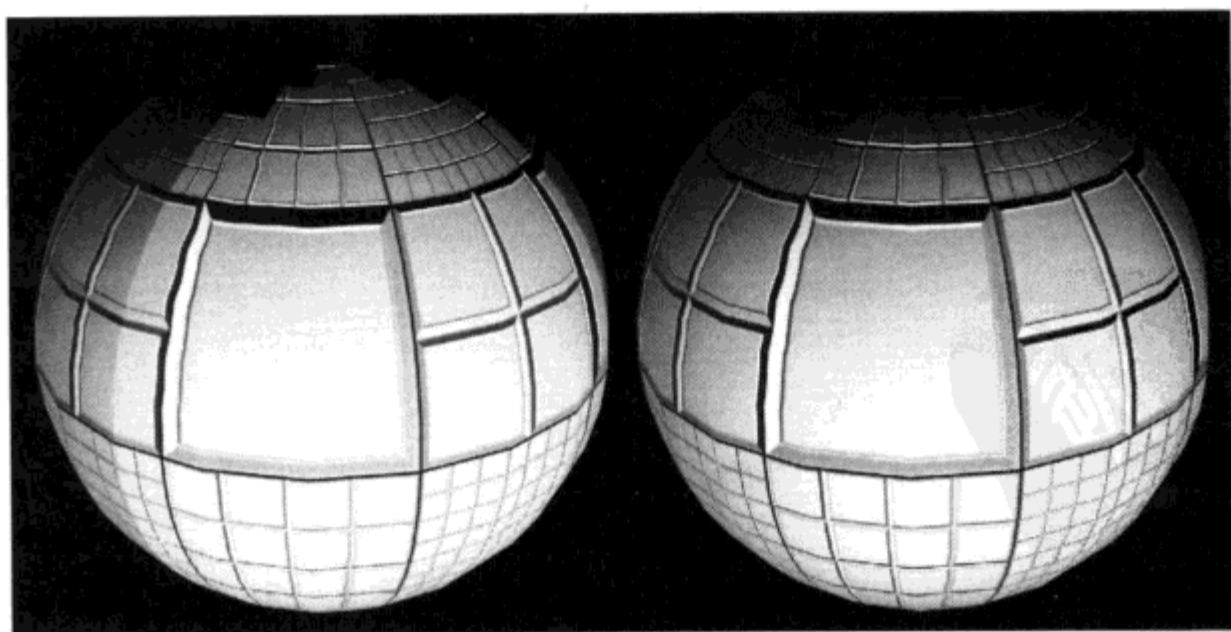


图 5.6.4 （左边）为逐像素光照操作使用凹凸贴图绘制了一个球体，并根据照射多边形的静态场景光的数量将它分成了多个网格模型。逐像素法线明显偏离面法线的地方，可以看到光照的不连续。

（右边）为了补偿这种差异，调整了漫射光照

5.6.4 结论

或许是为了优化或者是算法本身的需要，跟光照有关的操作常常使用多边形面法线来完成。本文论证了一个技术，它通过调整漫射光照的等式，可用于补偿由于基于表面法线的光照操作而产生的光照的不连续性。

5.6.5 参考文献

[Brennan02] Brennan, Chris, "Shadow Volume Extrusion Using a Vertex Shader," *Direct3D ShaderX: Vertex Shader Tips and Tricks*, Wordware Publishing, Inc., 2002.

Images provided by Eli Turner, ATI Research, Inc.



5.7 实时半调色法：快速而简单的样式化阴影

作者：Bert Freudenberg、Maic Masuch、Thomas Strothotte,
University of Magdeburg

E-mail: bert@isg.cs.uni-magdeburg.de,
masuch@isg.cs.uni-magdeburg.de,
tstr@isg.cs.uni-magdeburg.de

译者：刘永静

审校：谷超

本文介绍了半调色法，它是计算机游戏非图像真实性（nonphotorealistic）渲染样式的一种实现方式。这项技术只需要在公用硬件设备上使用传统的多纹理管道（multitexturing pipeline）。本文说明了如何为图像创建类笔墨绘图风格的半调色屏幕，以及如何使用合适的像素阴影硬件来快速实现半调色法渲染。

5.7.1 引言

在本文中，我们介绍了一个可以借鉴和适应从非交互式硬拷贝到实施环境的技术。类似的很多方法可作为灵感的来源。有些不够灵活 [Lake00] [Praun01]，有些难以理解 [Webb02]。

半调色法的原始形式是一个程序，它被用来打印那些只使用黑色墨水在灰色等级下的图像。它通过变更墨水点的尺寸来完成，因此，纸张面积就是着墨面积。从远处看，可察觉到一些色调。当一个美工使用笔墨绘图时，可以产生相似的效果，如果更多的线条被放置于区域中，就会呈现暗色。但是，雕刻或者木刻则使用的是另一种变化，他们通过调整线条的宽度来描述阴影的变化。所有这些样式都可以通过应用实时半调色法来重建。

同传统的半调色法相似，当一个特定的与像素相邻的图像亮度发生变化时，像素并没有失去光泽。而是更多的黑色像素被显示出来，剩下的像素则仍然保留白色，因此全部可被察觉到的亮度就减少了。相对于交互式环境，在对一个静态图片应用半调色法时，一个非常有意思的差异是，半调色屏幕（halftone screen）并不是要修理屏幕，更确切的应该是在3D空间中附上对象。另外，在半调色屏幕后面的对象将看起来像是在“游泳”，这就是众所周知的“淋浴门”效果。

5.7.2 原理

在半调色法中，决定视觉外观最多的基本因素就是半调色屏幕。这是一个包含临界值的灰度 (grayscale) 纹理。为了从一个特定的输入图像中创建一个半调色过的图像，每个像素的亮度被用来与相应的半调色屏幕的临界值相比较，根据比较的结果确定该像素是黑色还是白色 (见图 5.7.1)。如果 H 是半调色屏幕的临界值， L 是目标亮度，我们就可以用类 C 风格写出如下的临界函数。

$$H > (1-L) ? 1 : 0$$

这里， $(1-L)$ 表示“暗度”，它反映了墨水的负色彩模式 (subtractive color model)。

为了进行实时半调色法工作，我们需要一个半调色屏幕纹理和一个可执行临界操作的函数。半调色纹理指定了怎样的图像亮度应该被映射到黑白渲染中。它可能像本书中用于打印图像的规则点光栅 (the regular dot raster) 一样简单，或者也可能像图 5.7.3 中的砖块纹理一样复杂。

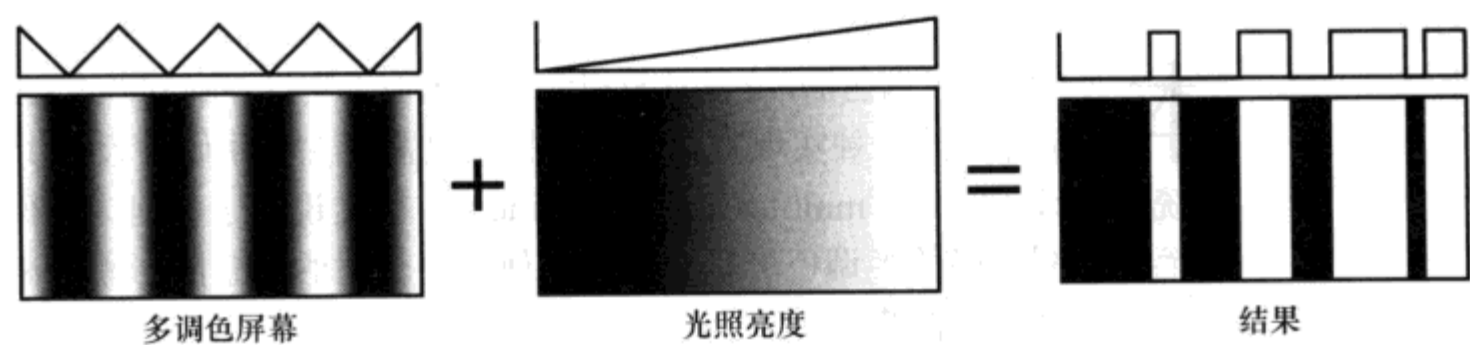


图 5.7.1 使用临界函数的传统半调色法。半调色屏幕中的每个值被用来同亮度值比较。如果半调色值大于暗度 (就是 1 减去亮度)，就产生一个白色像素；否则，就产生一个黑色像素

1. 创建半调色屏幕

半调色屏幕的创建可以通过程序或者手工实现。后者是更合人心意的，因为它更具有机动性。一个程序实现的半调色屏幕的例子是光滑的灰度条纹，如图 5.7.1 所示。这个结果是宽度取决于光照的线条，类似于木刻印刷风格 (见图 5.7.2)。其他用程序实现的屏幕可以被创建用于阴影[Lake00] [Praun01]。

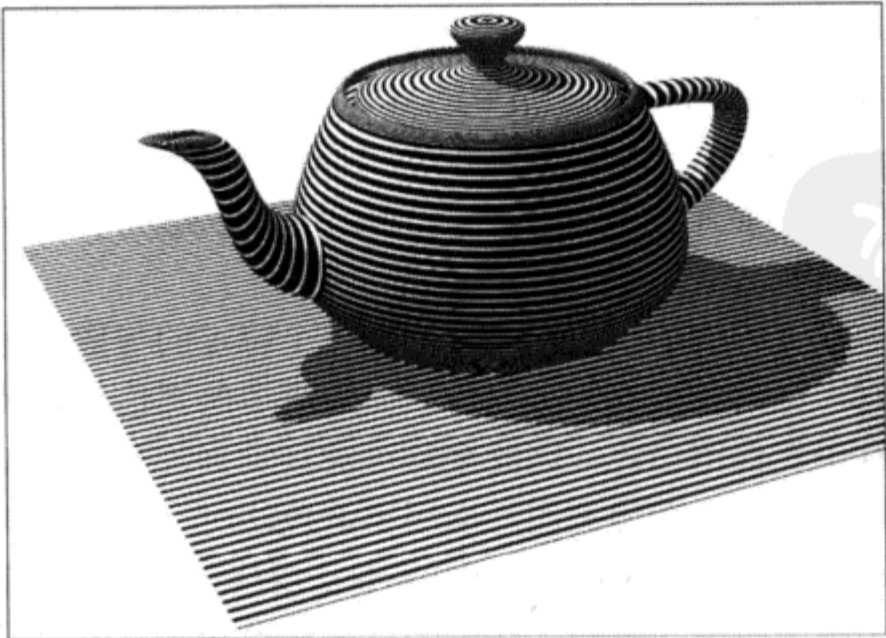


图 5.7.2 像图 5.7.1 中使用灰度条纹的半调色法，创建了一个类木刻样式。通过不同的线条宽度来传达阴影

为了手工创建半调色屏幕，我们开发了一个基于层的程序，以便达到好的效果（见图 5.7.3 中的最左边三幅图像）。每个图像都是在 Adobe Photoshop 中使用黑色在透明层上绘制出来的。当越来越多的层堆叠的时候，屏幕也变得越来越暗。这就是我们想要的用于半调色法的正确结果。当所有的层都被着色后，通过改变每一层到一个截然不同的灰色阴影中把它们组合成一个半调色屏幕。基本层保持黑色，而一些继承的层则被赋予更亮的灰色。合成过程的执行需要颠倒次序，因此黑色层应该在更亮层的顶部被绘制。结果图作为单一的 8 位灰度纹理被输出。

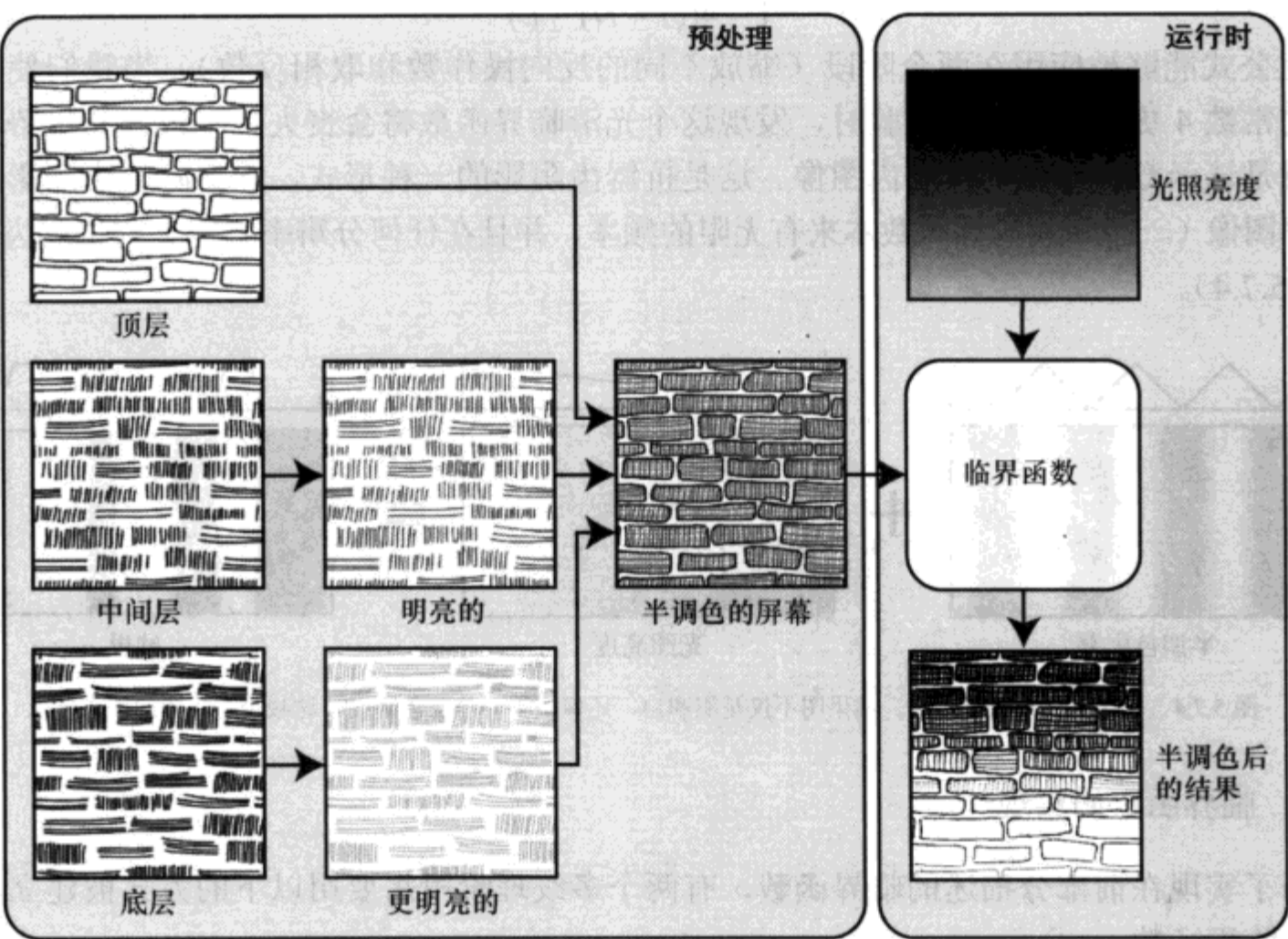


图 5.7.3 使用分层笔触的半调色法。通过使用笔触层的灰色等级对层的优先级进行编码，几个笔触层被组合成一个单一的半调色纹理。在运行时，为了产生半调色渲染，光照亮度被用来跟纹理的临界值相比较。结果，更多的笔触被显示在亮度更低的区域中

这个基于层的作品可以很好地为我们工作，但你也可以使用其他方法来创建半调色纹理。考虑半调色屏幕值的一个方法是每个像素为制图笔触（drawing stroke）编码一个“优先级”——黑色笔触具有更高的优先级，而且首先被显示，即使是在明亮的区域。低优先级的笔触（使用亮灰色描绘）只能显示在黑暗的区域，而半调色屏幕的白色部分也将在渲染中永远保持空白。

2. 限制像素阴影的临界函数

为了实现即时半调色法，必须在每个像素上使用临界函数来估值，此临界函数使用一个纹理和一个亮度值作为输入参数。这可以通过条件运算实现，根据比较结果输出黑色或白色

片断。更高级的像素阴影版本提供了一个可以使用的比较操作，而 OpenGL 中的最小公分母则用来支持 `texture_env_combine_ARB` 扩展。我们提出了一个用于光滑 (smooth) 临界函数的公式 (与正常用于临界值的锐利函数形成对比)，它只使用这个扩展提供的功能性。在下面的公式中，H 代表半调色屏幕，L 代表光照亮度。

$$1 - 4(1 - (H+L))$$

因为我们在这里是处理颜色，所以每项操作的结果都必须在 [0, 1] 之间。很不幸，直接执行这个公式需要三个纹理阶段 (求和、取相反数和缩放比例、再取相反数)，而一些普通的图形的边界只提供两个，重新做些小的调整，我们得出了以下的公式：

$$1 - 4((1 - H) - L)$$

此公式能够被应用在两个阶段 (缩放不同的反向操作数和取相反数)。当我们使用比等式中的常数 4 更大的值进行实验时，发现这个光滑临界函数将会丧失一个很好的优势，这个优势就是该函数所产生的抗锯齿图像。这是抗锯齿阴影的一种形式，它移除了由阴影产生的高频率图像 (一个锐利阶梯函数本来有无限的频率，并且在任何分辨率下都会产生锯齿) —— (见图 5.7.4)。

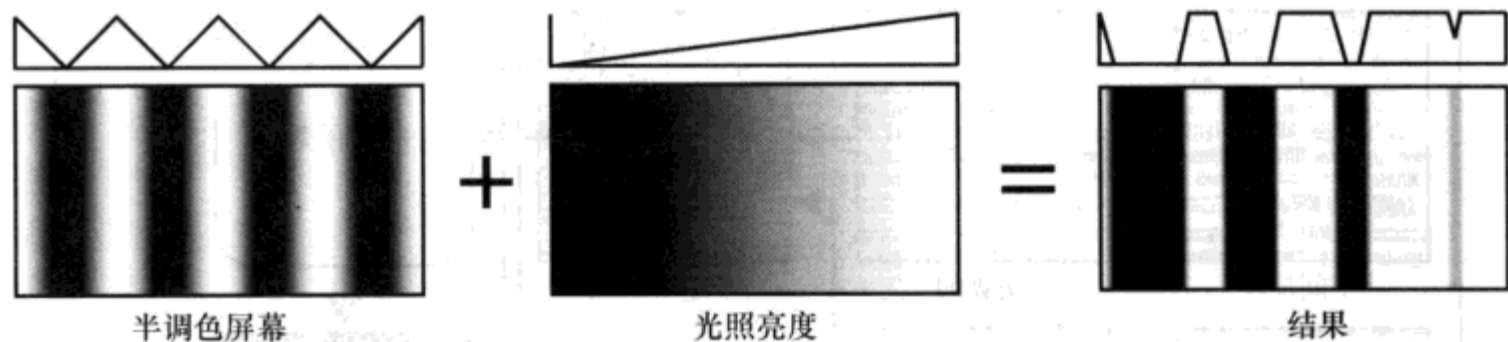


图 5.7.4 使用了光滑临界函数，结果图不仅是黑和白，还保留了一些灰色等级。这减少了阴影的锯齿现象

3. 临界函数的实现

为了实现在前部分描述的临界函数，有两个多纹理阶段需要用以下的方式被建立起来用于执行临界函数。

```
// 保存直线的宏
#define TexEnv(pname, param) \
    glTexEnv(GL_TEXTURE_ENV, pname, param)

// 阶段 0: 光滑临界函数 4*((1-L)-H)
glActiveTexture(GL_TEXTURE0);
TexEnv(GL_TEXTURE_ENV_MODE, GL_COMBINE);
TexEnv(GL_COMBINE_RGB, GL_SUBTRACT);
TexEnv(GL_SOURCE0_RGB, GL_PREVIOUS);
TexEnv(GL_SOURCE1_RGB, GL_TEXTURE);
TexEnv(GL_OPERAND0_RGB, GL_ONE_MINUS_SRC_COLOR);
TexEnv(GL_OPERAND1_RGB, GL_SRC_COLOR);
TexEnv(GL_RGB_SCALE, 4);

// 阶段 1: 反转函数 (1-previous)
glActiveTexture(GL_TEXTURE1);
```

```
TexEnv(GL_TEXTURE_ENV_MODE, GL_COMBINE);  
TexEnv(GL_COMBINE_RGB, GL_REPLACE);  
TexEnv(GL_SOURCE0_RGB, GL_PREVIOUS);  
TexEnv(GL_OPERAND0_RGB, GL_ONE_MINUS_SRC_COLOR);
```

第一个纹理阶段获得一个逐顶点光照数值 SOURCE0 和半调色纹理 SOURCE1。使用 ONE_MINUS_SRC_COLOR 输入贴图减去第一个操作数，并且所有的操作都被设置为 SUBTRACT。最后，将得到的结果再乘以 4。第二个纹理阶段仅使用一个操作数，并且使用它的相反数来代替它作为输入参数，再次使用输入贴图。

注意，你最好还是执行在纹理组合物 alpha 部分中的所有这些东西。相对于在其他地方使用的 RGB 部分，这是不耗资源的。如果图形硬件支持像素阴影，这个等式也可以在一个阴影程序中被实现。

5.7.3 实例的实现

为了示范将实时半调色法整合入传统的游戏引擎是如何的容易[Shark3D]，我们按照惯例将一个传统的纹理演示关卡[Requiem00]转换成笔墨的喜剧风格。在图 5.7.5 中，你可以看到我们修改前后的图片。

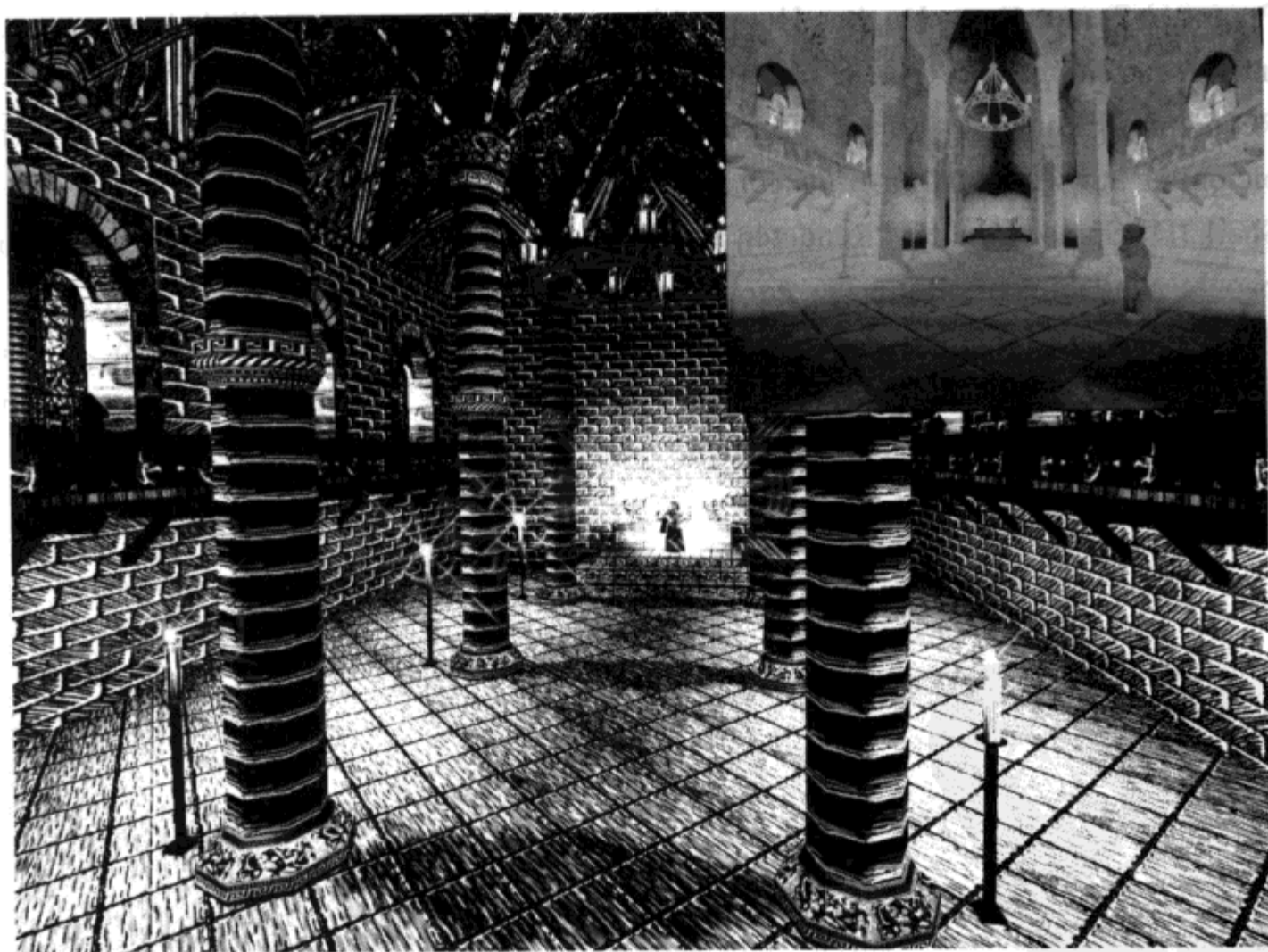


图 5.7.5 一个传统的普通纹理贴图与光照的关卡（见图中右上角）通过应用实时半调色法被转换成一个黑色的喜剧风格。注意阴影是如何通过显示的笔触数量的变化描述的

首先，纹理被转换成灰度（grayscale）。类似的，有色的光源被改变成灰度。一些纹理，

像天花板中或者有色的玻璃窗, 经过灰度转换后甚至没有进一步处理, 看起来就已经够好了。另一些纹理, 特别是墙和门, 则使用前面描述的层绘图技术, 使用类笔墨阴影风格绘制的。另外的改变包括用于蜡烛的发光效果的新纹理, 光照的亮度调整用于创建更清晰的阴影。在程序编写上, 只有阴影必须被重写用来实现临界函数。

5.7.4 结论

本文提出了实时半调色法, 这是一个适合游戏的简单而快速的技术, 用于实现非图像真实性的着色法。它可以为整个游戏或者特定的结果提供一个独特的外观。你可以想象一个游戏的入门介绍, 它的主要人物在阅读一部喜剧小说, 然后正确地绘制它。在实时图像中, 还有很多非图像真实性视觉风格至今还在被研究中。

5.7.5 参考文献

[Lake00] Lake, Adam, Carl Marshall, Mark Harris, and Marc Blackstein, "Stylized Rendering Techniques for Scalable Real-Time 3D Animation," *Proceedings of NPAR 2000, Symposium on Non-Photorealistic Animation and Rendering*, pp. 13–20.

[Praun01] Praun, Emil, Hughes Hoppe, Matthew Webb, and Adam Finkelstein, "Real-Time Hatching," *Proceedings of SIGGRAPH 2001*, pp. 581–586.

[Requiem00] Punkt im Raum, "Requiem: A Technology Study for the Shark 3D Engine," available online at www.punkt-im-raum.com/eng/projekte_requiem.shtml.

[Shark3D] Spinor GmbH, "Renderer and Shader Architecture," Shark3D engine technology white papers, available online at www.shark3d.com/goto/technology_render.html.

[Webb02] Webb, Matthew, Emil Praun, Adam Finkelstein, and Hugues Hoppe, "Fine Tone Control in Hardware Hatching," *Proceedings of NPAR 2002, International Symposium on Non Photorealistic Animation and Rendering*, pp. 53–58.



5.8 在 3D 模型中应用团队色的各种技术

作者: Greg Seegert, Stainless Steel Studios

E-mail: gseegert@alum.wpi.edu

译者: 刘永静

审校: 谷超

在任何多角色游戏中, 无论是 AI 还是人工控制, 在三维模型上应用引人入胜的团队色 (team color) 都是一个很重要的技术。本文探索了几种可应用到任意三维模型上的团队色的技术, 详细地解释了每种技术的实现, 并且讨论了每种方法的优点和缺点。不同的应用团队色到三维模型中的方法会影响美工组如何为你的游戏创建素材。因此, 关于在游戏中选择一个适当的团队色技术, 在项目中尽可能早的做出见多识广的决定显得非常重要。做出一个选择最适合你的项目目标的技术的决定能够节省大量花在美工上的时间。

5.8.1 什么是团队色?

团队色是一种在很多类型的游戏中经常使用的技术。团队色最基本的用途是快速地区别三维游戏中的各种团队。在避免美工进度表上出现不必要的紧张方面和为游戏中的每个团队创建自定义模型或者纹理上的内存浪费方面也很有帮助。例如, 大多数即时策略游戏使用这种技术来区别不同玩家所拥有的军队和属性。本文所讨论的技术不只是限于区别团队, 它们还可用于其他目的, 比如在竞赛游戏或者太空船模拟游戏中自定义玩家的交通工具。

5.8.2 团队色的算法

有几种算法可将团队色应用到三维模型中, 每种算法都有各自的优点和缺点。我们将看到以下几个算法: 为每个团队色使用独一无二的纹理; 为模型多边形着色以形成适当团队色; 使用多纹理进行纹理遮挡; 使用多种途径进行纹理遮挡; 高级像素阴影效果。

1. 独一无二的纹理

这个技术是最强劲的实现方式。它简单地游戏中想要支持的每个团队色创建一个独一无二的纹理。这样一来, 它几乎不需要编程, 但是却需

要花费最高水平的美术调节 (artistic control)。这种技术的主要缺点是需要消耗大量内存。

优点

- 几乎不需要编程就可以实现。
- 需要美术调节的最高水平才能得到结果纹理。

缺点

- 大量的纹理内存被浪费。
- 对美工组来说, 纹理改变的维护是一件困难的事。
- 如果为了减轻内存的约束而牺牲纹理分辨率的话, 就可能出现低品质纹理。

实现

不需代码实现。

2. 多边形着色

多边形着色就是对在网格模型上美工设计好的多边形根据玩家团队的颜色进行着色的方法。这是使用在很多游戏中的相当普遍的方法, 但是有几个严重的限制。

优点

- 对任何数量的团队不需要额外的纹理内存。

缺点

- 美术调节有限。美工被迫要设计一个巨大的多边形区域作为团队色多边形, 这就导致了模型缺乏吸引力。这项技术进一步压缩了艺术品的创作, 因为美工必须避免在这些多边形上使用颜色, 以便可以显示正确的团队色。
- 影响渲染性能。网格模型必须使用两个单独的绘制调用和它们之间渲染状态的变化来绘制, 这就限制了定量的机会, 潜在地影响了渲染性能。然而, 通过聪明地使用顶点阴影来避免这一点是完全有可能的。

DirectX 实现

```
/* 将材料设置为纯白色 */
D3DMATERIAL9 theMaterial;
theMaterial.Diffuse.r = theMaterial.Ambient.r = 1.0F;
theMaterial.Diffuse.g = theMaterial.Ambient.g = 1.0F;
theMaterial.Diffuse.b = theMaterial.Ambient.b = 1.0F;
theMaterial.Diffuse.a = theMaterial.Ambient.a = 1.0F;
mD3DDevice->SetMaterial(&theMaterial);

/* 纹理阶段 0: 正常地设置纹理 */
mD3DDevice->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE);
mD3DDevice->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
mD3DDevice->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
mD3DDevice->SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_MODULATE);
mD3DDevice->SetTextureStageState(0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE);
mD3DDevice->SetTextureStageState(0, D3DTSS_ALPHAARG2, D3DTA_DIFFUSE);

/* 纹理阶段 1: 使之不可用 */
```



```
mD3DDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_DISABLE);
mD3DDevice->SetTextureStageState(1, D3DTSS_ALPHAOP, D3DTOP_DISABLE);

/* 只绘制模型的非团队色多边形 */
mD3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST ...

/* 为材料设置团队色（本例使用黄色）*/
theMaterial.Diffuse.r = theMaterial.Ambient.r = 1.0F;
theMaterial.Diffuse.g = theMaterial.Ambient.g = 1.0F;
theMaterial.Diffuse.b = theMaterial.Ambient.b = 0.0F;
theMaterial.Diffuse.a = theMaterial.Ambient.a = 1.0F;
mD3DDevice->SetMaterial(&theMaterial);

/* 只绘制团队色多边形 */
mD3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST ...
```

3. 纹理遮挡，多纹理

这项技术包括通过使用模型纹理的 **alpha** 通道来允许美工“画”纹理中的团队色区域。它为美工提供了极大的自由度，低纹理内存消耗，在支持硬件上实现高性能。

优点

- 高水准的美术调节。美工可以在纹理的 **alpha** 通道中绘制团队色区域。这就允许美工创建复杂的图案、模式，然后把它们同团队色一起完全整合入模型中。例如，一辆军用车辆可能有点褪色和擦伤的痕迹，指定的数字图案被光滑地掺入到伪装的喷绘图案中，所有的这些都可以使用团队色。

- 更少的纹理内存需求。由于这项技术利用了纹理的 **alpha** 通道，不需要额外的纹理来指定任何数量的团队色。如果模型并不需要高分辨率的团队色，那么就可以使用 1 位 **alpha** 通道，外加各种各样的可压缩 1 位 **alpha** 通道的纹理压缩算法，此技术在内存消耗上提供了更多的优势。

- 提高性能。在那些不需要使用 **alpha** 通道来解决透明度的模型上，整个模型就在一个绘制调用中被渲染。此外，支持多纹理技术所需的硬件将受益于高性能。

缺点

- 硬件兼容性。某些图形硬件可能不能支持所需的纹理场所（**texture stage**）和操作。这导致了此技术不能用于这些系统，如果团队色是游戏操作的鉴定特征，那么就会出现严重的问题。

- 排除了那些可解决透明度的 **alpha** 通道的使用。使用 **alpha** 通道来指定团队色意味着 **alpha** 通道也不能用于透明度。这个问题的一个主要工作就是需要美工指定网格模型上哪些多边形使用这项技术，哪些多边形不使用这项技术。然后就可以使用该技术来渲染那些需要使用这项技术的多边形，而另一些多边形则可以被正常的渲染。然而，由于现在模型必须在多重绘制调用中被绘制，因此就可能影响性能。如果有必要，就只需要通过使用被指定为透明的纹理来限制这些性能上的负面影响。不过，在最近的那些可以支持许多多纹理阶段或者像素阴影的硬件上，这个不是问题。

DirectX 实现

```
/* 将团队色放入 TFACTOR（本例使用黄色）*/
```

```

mD3DDevice->SetRenderState(D3DRS_TEXTUREFACTOR,
D3DCOLOR_COLORVALUE(1.0F, 1.0F, 0.0F, 1.0F));

/* 纹理阶段 0: 在纹理颜色间混合
 * 并且 TFACTOR 依赖于 alpha 纹理
 * 我们仅为 alpha 使用漫射 */
mD3DDevice->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_BLENDTEXTUREALPHA);
mD3DDevice->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
mD3DDevice->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_TFACTOR);
mD3DDevice->SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1);
mD3DDevice->SetTextureStageState(0, D3DTSS_ALPHAARG1, D3DTA_DIFFUSE);
mD3DDevice->SetTextureStageState(0, D3DTSS_ALPHAARG2, D3DTA_DIFFUSE);

/* 纹理阶段 1: 使用漫射调节当前纹理来获得照明
 * 不使用 alpha 做任何事 */
mD3DDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_MODULATE);
mD3DDevice->SetTextureStageState(1, D3DTSS_COLORARG1, D3DTA_CURRENT);
mD3DDevice->SetTextureStageState(1, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
mD3DDevice->SetTextureStageState(1, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1);
mD3DDevice->SetTextureStageState(1, D3DTSS_ALPHAARG1, D3DTA_CURRENT);
mD3DDevice->SetTextureStageState(1, D3DTSS_ALPHAARG2, D3DTA_CURRENT);

/* 纹理阶段 2: 使之不可用 */
mD3DDevice->SetTextureStageState(2, D3DTSS_COLOROP, D3DTOP_DISABLE);
mD3DDevice->SetTextureStageState(2, D3DTSS_ALPHAOP, D3DTOP_DISABLE);

/* 渲染模型 */
mD3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST ...

```

4. 纹理遮挡, 多途径

通过使用一个多途径解决方案, 对多纹理技术 (multitexturing technique) 来说, 有可能得到同样的结果。这种技术需要渲染部分网格模型两次。第一次, 网格模型被正常地渲染。在第二步期间, 网格模型被着色上想要的团队色, 并且反转了混合模式, 以致在纹理的 alpha 通道中, 网格模型只被渲染成透明。

优点

- 良好的硬件兼容性。这个技术的优点跟那些多纹理解决方案是同样的, 新增的一个优点是, 能够很好地同各种各样的图形硬件兼容。

缺点

- 对性能的影响。这个技术将对性能产生影响, 因为网格模型中的多边形需要被渲染两次。此外, 渲染状态的改变需要用两种途径对整个网格模型进行渲染。如果美工能够指定需要被 team color 激活的最少数量的多边形, 那么可以将其他需要被渲染的多边形的数量减到最少。

DirectX 实现

```
/* 将材料设置为纯白色 */
```

```
D3DMATERIAL9  theMaterial;
theMaterial.Diffuse.r = theMaterial.Ambient.r = 1.0f;
theMaterial.Diffuse.g = theMaterial.Ambient.g = 1.0f;
theMaterial.Diffuse.b = theMaterial.Ambient.b = 1.0f;
theMaterial.Diffuse.a = theMaterial.Ambient.a = 1.0f;
mD3DDevice->SetMaterial(&theMaterial);

/* 设置正常的混合 */
mD3DDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
mD3DDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);

/* 纹理阶段 0: 正常地设置纹理
 * 只让 alpha 漫射通过 */
mD3DDevice->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE);
mD3DDevice->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
mD3DDevice->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
mD3DDevice->SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG2);
mD3DDevice->SetTextureStageState(0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE);
mD3DDevice->SetTextureStageState(0, D3DTSS_ALPHAARG2, D3DTA_DIFFUSE);

/* 纹理阶段 1: 使之不可用 */
mD3DDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_DISABLE);
mD3DDevice->SetTextureStageState(1, D3DTSS_ALPHAOP, D3DTOP_DISABLE);

/* 绘制所有模型的多边形 */
mD3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST ...

/* 转换混合模式 */
mD3DDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_INVSRCALPHA);
mD3DDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_SRCALPHA);

/* 为材料设置团队色 (本例中使用黄色) */
theMaterial.Diffuse.r = theMaterial.Ambient.r = 1.0f;
theMaterial.Diffuse.g = theMaterial.Ambient.g = 1.0f;
theMaterial.Diffuse.b = theMaterial.Ambient.b = 0.0f;
theMaterial.Diffuse.a = theMaterial.Ambient.a = 1.0f;
mD3DDevice->SetMaterial(&theMaterial);

/* 纹理阶段 0: 只为颜色使用漫射, 为 alpha 使用纹理 */
mD3DDevice->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_SELECTARG2);
mD3DDevice->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
mD3DDevice->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
mD3DDevice->SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG1);
mD3DDevice->SetTextureStageState(0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE);
mD3DDevice->SetTextureStageState(0, D3DTSS_ALPHAARG2, D3DTA_DIFFUSE);

/* 纹理阶段 1: 使之不可用 */
mD3DDevice->SetTextureStageState(1, D3DTSS_COLOROP, D3DTOP_DISABLE);
mD3DDevice->SetTextureStageState(1, D3DTSS_ALPHAOP, D3DTOP_DISABLE);
```

```
/* 还是只绘制团队色多边形 */
mD3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST ...
```

5. 高级纹理遮挡，像素阴影的实现



在渲染团队色中，像素阴影的使用提供了很多提高团队色技术的机会。这是一个相对简单的技术，它通过在像素阴影中仿效纹理遮挡技术来实现，但是像素阴影也提供了很多增强效果的机会。潜在的可提高区域包括每个模型使用多重团队色或者细微的色调偏移，将一个不同的光照模型应用到团队色区域，依靠视觉的特殊效果，在多纹理应用程序中更多的弹性。随书附带的 CD-ROM 上的代码实例包括一个通过 second texture 混合了两种团队色的技术，而这个 second texture 的 UV 坐标改变是基于各项异性的反射系数的 (anisotropic reflectance)。镜面条件也被加入到团队色区域中，用来增加可感知到的团队色的“亮光”，并且用来证明不同的光照。

优点

- 最大化实现灵活性和应用特殊效果的能力将导致使用固定的函数管道来实现变得非常困难或者不可能。

缺点

- 硬件的兼容性。如果一些目标图形硬件不支持像素阴影，将需要使用一个转换后的技术。

实现



- 完整的顶点和像素阴影列表请参见随书附带的 CD-ROM。

5.8.3 一个实际的例子

在一个不锈钢工作室，我们决定对第一级标题 *Empire Earth* 中使用的团队色技术加以改进。*Empire Earth* 使用多边形着色算法，这个算法适合低多边形模型和目标平台。然而，在开发下一个标题 *Empires: Dawn of the Modern World* 期间，这个古老的多边形着色技术很快变成了明显不能满足大多数的 high detail 模型。艺术调节的缺乏导致了美工成员的挫折感，新模型的总质量退化了。用来弥补这种情形的几种算法的原型在这篇文章中所讨论的一些技术中的产生了。

1. 对多边形着色技术的支持

尽管它有缺点，但对我们来说，继续支持多边形着色技术是很重要的。新美术素材来临的时候，已经存在的素材需要一个方法来渲染团队色。而且，多边形着色技术被证明可以完全适应某些模型，例如建筑物和一些交通工具，因为它们斑驳的外形没有必要用其他复杂的方法实现。

2. 对多纹理纹理遮挡技术的支持

纹理遮挡技术被证实是 *Empires: Dawn of the Modern World* 的理想技术。虽然这个技术可

以为美术组提供极好的美术调节，但是我们还是发现了潜在的问题。额外的 alpha 通道导致了纹理尺寸的增长。通过在适当的地方使用 1 位 alpha 通道和 DXTC (DirectX 纹理压缩格式) 可以解决这个问题。另一个障碍包括多纹理技术的使用。采用多纹理排除了这项技术影响了兼容性，导致我们的程序无法在所有的显卡上正常工作。使用多途径技术可以成功地解决硬件兼容性的问题。

3. 对多途径纹理遮挡技术的支持

为了支持游戏可玩性，评测团队色技术，这成为解决通过多纹理技术所暴露的不兼容性的问题的必要手段。通过多途径模拟多纹理技术的效果被证实是一个高效的工作方法。在图形硬件不能够支持多纹理技术的情况下，渲染引擎将回退到使用多途径解决方案。

4. 对性能的影响

我们花费几步来解决性能的影响。就像前面提到的一样，只要有可能减少纹理内存所覆盖 (footprint) 的影响，美术组就会被鼓励去使用 1 位 alpha。而且，当在一个单一的绘图调用中渲染模型的时候，多纹理技术有可能有助于达到最佳性能。然而，这需要模型在不能够支持必要的多纹理阶段的图形硬件上被渲染两次。因此，我们决定，团队色多边形将被限制到模型中的多边形总数的一个固定的百分比。虽然这需要在两个单独的绘图调用中使用团队色来渲染所有的模型，但是它同时减少了多途径回滚算法对性能的影响。

5.8.4 光盘中的内容



附带的代码包括一个 DirectX 9.0 应用程序，它显示了所有在这里已经讨论过的技术，这个程序建立在 Stainless Steel Studios 的游戏 *Empires: Dawn of the Modern World* 的模型上。实例代码也包括了一个使用顶点和像素阴影的高级技术。

5.8.5 结论

本文中，我们探索了不同的可用于渲染三维模型的团队色技术。每种技术都有自己的强项和弱项，这些都应该在项目中及早评估，以便为特定的游戏决定一个最好的解决方案。一个经过仔细选择的团队色方法可以挽回无数美术和编程的时间，同时提高你游戏中的模型的全面品质，并且可以使用更少的模型来支持大量的玩家。

5.9 快速的棕褐色色调转换

作者: Marwan Y. Ansari, ATI Research, Inc.

E-mail: mansari@ati.com

译者: 刘永静

审校: 谷超

通 过后处理实现一个样式化的外观在游戏中日益流行, 将图像从 RGB 空间转换成棕褐色色调就是一个这样的技术。棕褐色色调是一种被用来使图像呈现出老化或者陈旧感觉的色彩空间。从 RGB 到棕褐色的色彩转换通常通过映射一个 RGB 色彩到查询表中来完成的。虽然这个技术简单并且有效, 但是我们还是发现了一个更简单和更快速的方法可以完成相同的转换, 它只需要一些 pixel shader 指令, 不需要表查询。

要完成这个转换, 我们需要先把 RGB 实例从输入图像转换到 YIQ 空间, 对其进行处理, 然后再把它转换回 RGB 空间。本文不仅描述了常规的转换方法, 而且描述了优化的方法。

5.9.1 背景

YIQ 空间同 YUV 空间相似, 有时用于电视广播[Jack93]。它被定义为 R、G、B 值的线性组合。色彩通过简单的矩阵相乘, 就可以从原来的色彩空间转换到 YIQ 空间, 也可以从 YIQ 空间转换回原来的色彩空间, 如[Foley96]中所示。

将 RGB 实例转换到 YIQ 的矩阵为:

$$M = \begin{pmatrix} 0.299 & 0.587 & 0.144 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{pmatrix}$$

M 的第一行用来计算输入的 RGB 实例的亮度, 就是 YIQ 中的 Y 部分。第二、第三行分别是 I 和 Q 的编码[Foley96]。

从 YIQ 转换回到 RGB 的矩阵是 M 的反过程:

$$M' = \begin{pmatrix} 1.0 & 0.956 & 0.620 \\ 1.0 & -0.272 & -0.647 \\ 1.0 & -1.108 & 1.705 \end{pmatrix}$$

5.9.2 常规的方法

在我们讨论优化之前先让我们讨论一下这种转换的一种蛮力方法。

- (1) 通过一个矩阵同 M 相乘, 将 RGB 实例转换为 YIQ。
- (2) 分别使用 0.2 和 0.0 替换 I 和 Q 部分。
- (3) 通过一个矩阵同 M' 相乘, 将 YIQ 的值转换回 RGB。

这项操作最有趣的一部分就是使用 0.2 和 0.0 来替换 I 和 Q 。为了真正搞明白这里发生了什么, 让我们来看看从 YIQ 到 RGB 的转变。 $M'[YIQ]^T$ 的操作被扩展为:

$$R' = I + 0.956 * I + 0.620 * Q$$

$$G' = I + 0.272 * I + 0.647 * Q$$

$$B' = I + 1.108 * I + 0.705 * Q$$

使用 0.2 替换 I 并使用 0.0 替换 Q , 等式被简化并重写为:

$$R' = Y + 0.191$$

$$G' = Y - 0.054$$

$$B' = Y - 0.221$$

在这里我们可以看到, 这个操作使用计算过的亮度偏差简单地替换了色彩通道, 产生了一个棕褐色色调图像。

数值 0.2 和 0.0 是通过实验确定的, 在不同的应用程序中, 其他值看起来可能会更好。要不断实验, 直到程序员或者美工找到适合他们应用程序的最佳常数为止。

很明显, 这个方法也可被用来将灰度图像直接转换为棕褐色色调。由于程序中最重要的是输入参数是亮度, 因此解决像这样的转换, 此方法是最理想的。

5.9.3 优化

由于上述重写后的色彩空间转换只是计算后的亮度的简单偏移, 因此很容易看到如何通过调整一些指令来优化这个算法。由于需要做的工作只是计算 RGB 实例的亮度值, 然后加上一个常数, 因此我们可以将工作减少到只需要两项运算: 点乘积运算和加法运算。这比为了转换, 让一个矩阵来回地乘以 YIO 空间, 或者存储一个查询表来得更高效。

在这里我们使用 DirectX 9 HLSL 语言来实现优化后的算法。

```
sampler inputImage;

float4 Sepia_Optimized(float2 tc : TEXCOORD0) : COLOR
{
    float Y;
    float4 c, currFrameSample;
    float3 IntensityConverter= {0.299, 0.587, 0.114};
    float4 sepiaConvert = {0.191, -0.054, -0.221, 0.0};

    // get sample
    currFrameSample = tex2D(inputImage, tc);

    // get intensity value (Y part of YIQ)
    Y = dot(IntensityConverter, currFrameSample);

    // Convert to Sepia Tone by adding constant
```

```
    c = Y + sepiaConvert;

    return c;
}
```

就如我们想象的一样，这个 HLSL 代码只编译成两个算法运算：

```
ps_1_1
def c0, 0.299, 0.587, 0.114, 0
def c1, 0.191, -0.054, -0.221, 0
tex t0
dp3 r0, c0, t0
add r0, r0, c1
```

5.9.4 结论

在这篇文章中，我们说明了一个彩色图像到棕褐色色调的转换，由于通过调整两个 pixel shader 操作就可以实时完成这种转换，因此它并不需要一个查询表。使用这种方法在仍可以给出一个可接受的结果的同时可以减少内存和时间的消耗。

5.9.5 参考文献

[Foley96] Foley, James, Andries van Dam, et al., *Computer Graphics: Principles and Practice, Second Edition*, Addison-Wesley, 1996.

[Jack93] Jack, Keith, *Video Demystified: A Handbook for the Digital Engineer*, HighText, 1993.



5.10 使用场景亮度采样实现动态的 Gamma

作者: Michael Dougherty 和 Dave McCoy

E-mail: mdougher@hotmail.com,david.mccoy@comcast.net

译者: 刘永静

审校: 谷超

人的眼睛经常采样可见光的亮度并且会相应地放缩瞳孔。这种变化的灵敏性可以允许眼睛在一个非常宽的亮度条件范围内工作, 比如从黑暗到光明。通过采样帧缓冲和调节基于数据分析的输出 gamma, 本文中的技术要点能够模拟这种类型的灵敏性变化, 从而使视频 (video) 有限的动态范围得到更好地使用。同样的技术还可以模拟当眼睛响应变化的、高度动态范围的照明时所发生的大量的视觉现象。

5.10.1 光照系数

由于光照传播系数是一个乘积 (光的颜色 \times 材料的颜色), 在游戏图像中, 白色或者连贯的明亮像素倾向于稀疏。任何不同于纯粹的白光源的光源照在纯粹的白色材料上将导致一个比白色暗一点的阴影。不管如何明亮的光源, 一个灰色的材料永远不可能被绘制得比本身更亮一点。最新一代的实时图形硬件考虑了更高精度的阴影计算, 但是在当前固定精度为 24 位的硬件上普遍存在的基于光照的传播系数并没有考虑亮度超过 1.0 的光源的表达式。

所有这些的结果就是大多数游戏都不能使用视频的已经非常有限的动态范围的全部范围, 并且输出结果跟理想状态相比, 是更加暗、更加差的。本文阐述了如何通过模拟眼睛对光照环境改变的即时反应来弥补这些不足。

5.10.2 有限的动态范围

图 5.10.1 是使用一个相当典型的亮度分布捕捉的一个屏幕。图 5.10.2 是图像的亮度 (明亮) 柱状图, 它显示这个图像没有完全使用有效的动态范围, 并且图像显然偏向更暗的值。就像在很多当前的游戏中一样, 输出结果的外观是相当暗的, 仿佛有一个无光膜覆盖在我们的监视器屏幕上一样。由此推测, 一个明亮的有阳光照射场景看起来就好像和华盛顿的一个很普通的下雨天一样沉闷。

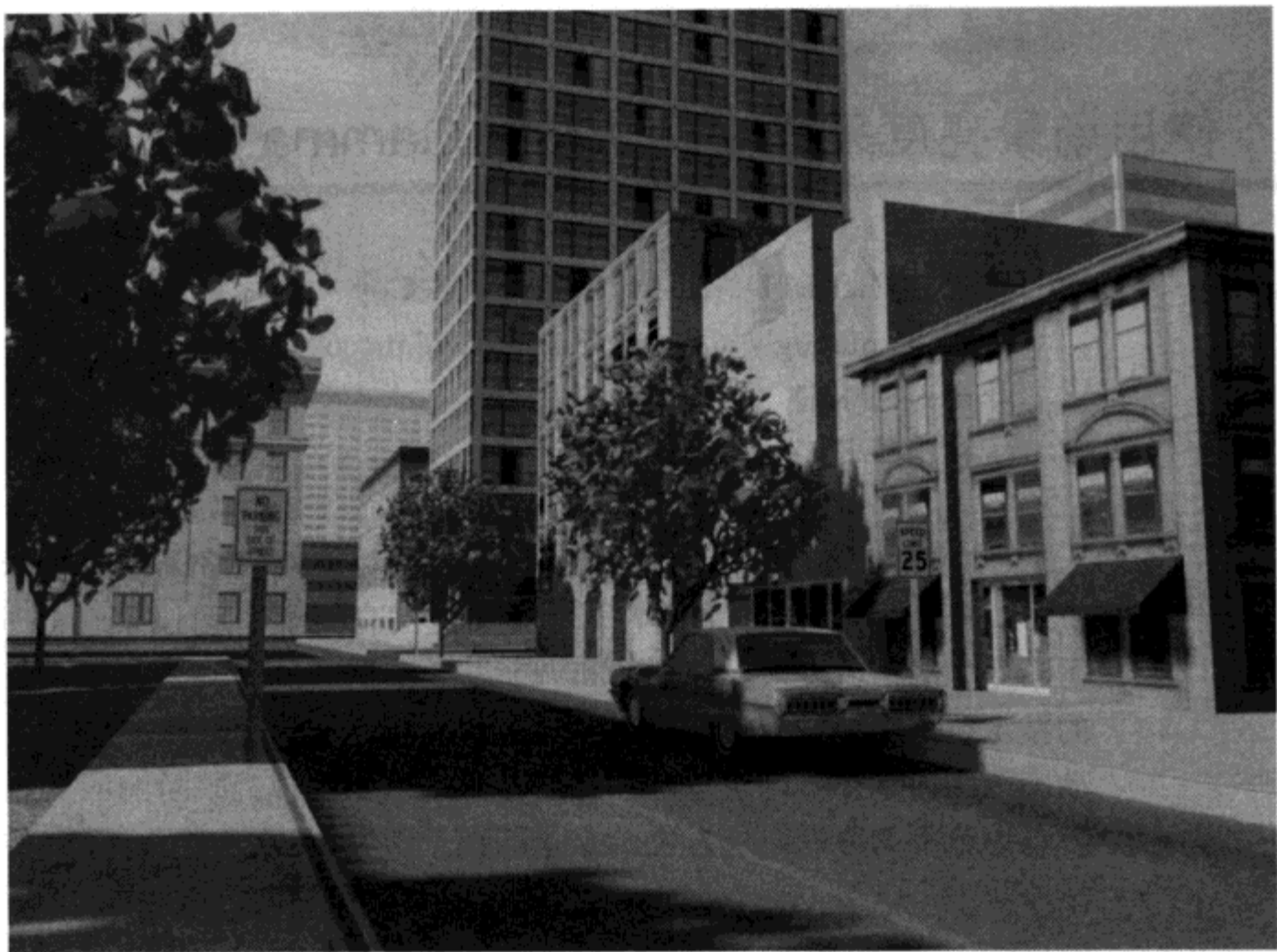


图 5.10.1 一个典型的使用光照系数的户外场景

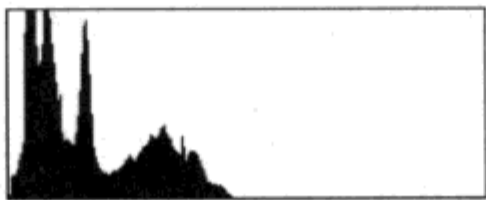


图 5.10.2 图 5.10.1 的亮度柱状图

如果视频监视器增加大约 10 000 倍的亮度，我们的游戏就可以复制我们的眼睛所能够察觉到的亮度的全部范围，那么我们的眼睛就可以对游戏图形起反应，就像它们对现实世界起反应一样。不幸的是，视频监视器不能够达到这个挑战。视频屏幕所能够制造的最明亮的光仅比最黑暗的大约明亮 100 倍，因此我们不得不使用比一个 50W 灯泡稍微暗一点的输出值来模拟眼睛对不同照明环境的反应。

5.10.3 图像的优化

一般而言，如果眼睛观察很少亮度或者没有亮度的光，它就会放大瞳孔来接纳更多的光。类似地，我们可以分析一个图像，如果我们发现它包含很少或者没有亮的像素，那么我们可以适当地放大输出值的亮度。这就使我们有能力通过使用已经对显示的动态范围优化过的输出值，对游戏中的不同亮度等级进行反应。非常有趣的是，这实际上是使美工和设计者能够自由地使用暗度等级和亮度等级来开发环境，而不需要太关心输出值的适宜性。除了创建令

人高兴的更和谐的、更饱和的、更高的对比的图像之外，这个后处理步骤也导致了图像中最亮的像素被推进到非常高亮度的等级。这导致了现实中的过分饱和，它是当明亮的灯光照亮浅颜色的表面的时候我们看到的一种典型的现象。图 5.10.3 显示的是图 5.10.1 在进行动态范围扩张后的图像数据。这种扩张后图像的柱状图如图 5.10.4。

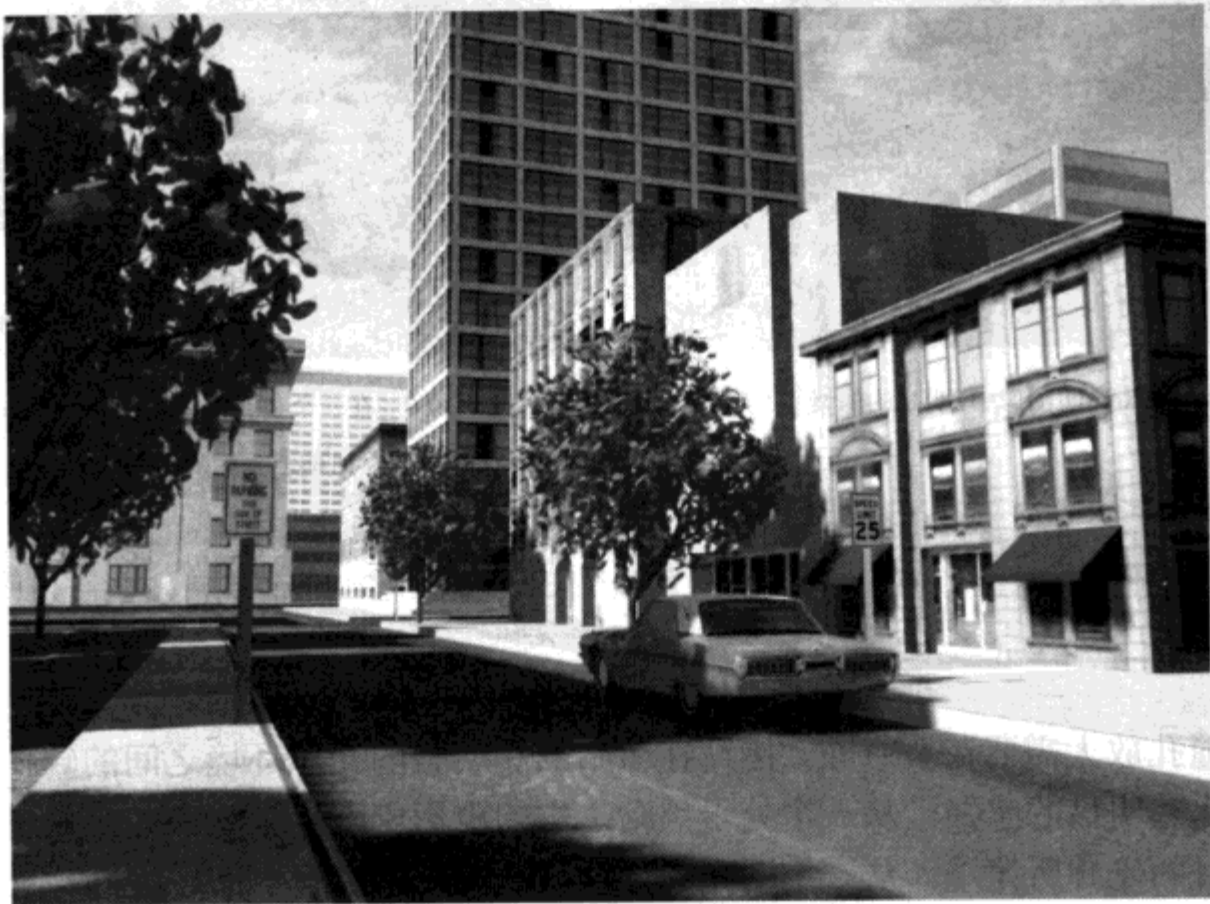


图 5.10.3 使用动态 Gamma 调整后的系数被照亮的场景

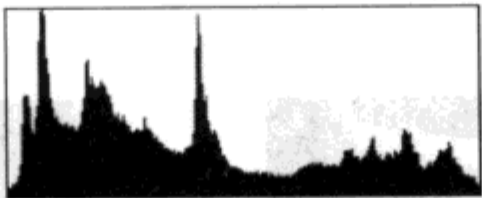


图 5.10.4 图 5.10.3 的亮度柱状图

正像我们的眼睛永远不能容纳足够的光来使一个没有月亮的晚上像白天一样亮，我们的算法执行所得到的调节的数量可以被任意的限制来避免不期望的结果。

5.10.4 易变的光灵敏度

在相差很大的不同的光照环境下，使用最有可能的动态范围不只可以生产图像，眼睛的灵敏度变化导致了大量有趣的现象。例如：我们在不同的全照明环境下对光源的感知是完全不同的。在晚上令人炫目的闪光灯在白天几乎不能够让人感知。我们的算法制造了相同的效果。在主要为黑暗的图像中，明亮一点的像素的亮度要比它们在普通的明亮图像中更亮。

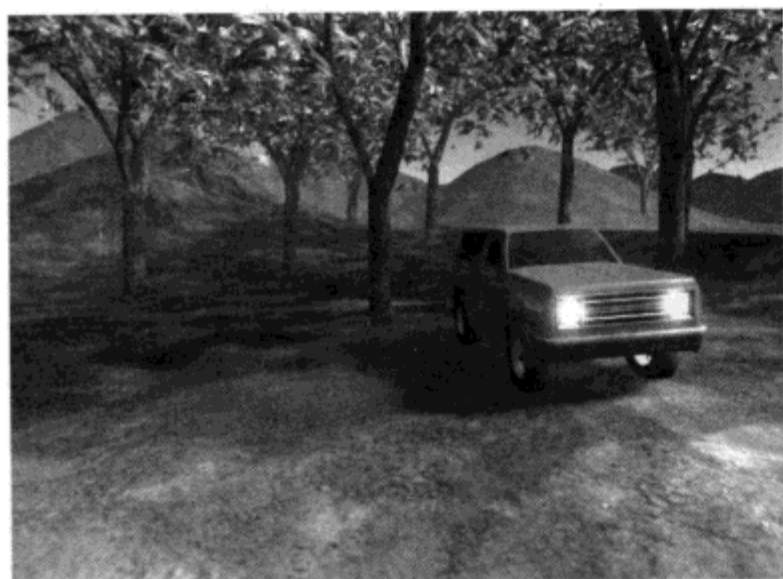


图 5.10.5 在一个一般明亮的图像中, 明亮的像素不是特别引人注目



图 5.10.6 在一个主体为黑暗的图像里, 同样明亮的透镜喇叭明亮地照亮了显得更亮的表面

5.10.5 转换

由于使瞳孔放大需要花费时间。因此在黑暗的和明亮的光照环境之间的转换可以导致我们对光照变化有明显的感知。从一个处于黑暗之中的电影剧院走到一个完全白昼的地方, 人们就会被炫目的光辉所笼罩。从一个明亮的环境移动到一个暗一点的环境同样产生了陷入一片漆黑的现象, 直到我们的眼睛调整好。通过调节我们的技术的调节速度可以产生相同的效果。这些效果如同图 5.10.7 到 5.10.11 所示。在每种情形中, 左边的一帧显示的是未被改变的图片, 右边一帧显示的是经过 gamma 调节后的图像。

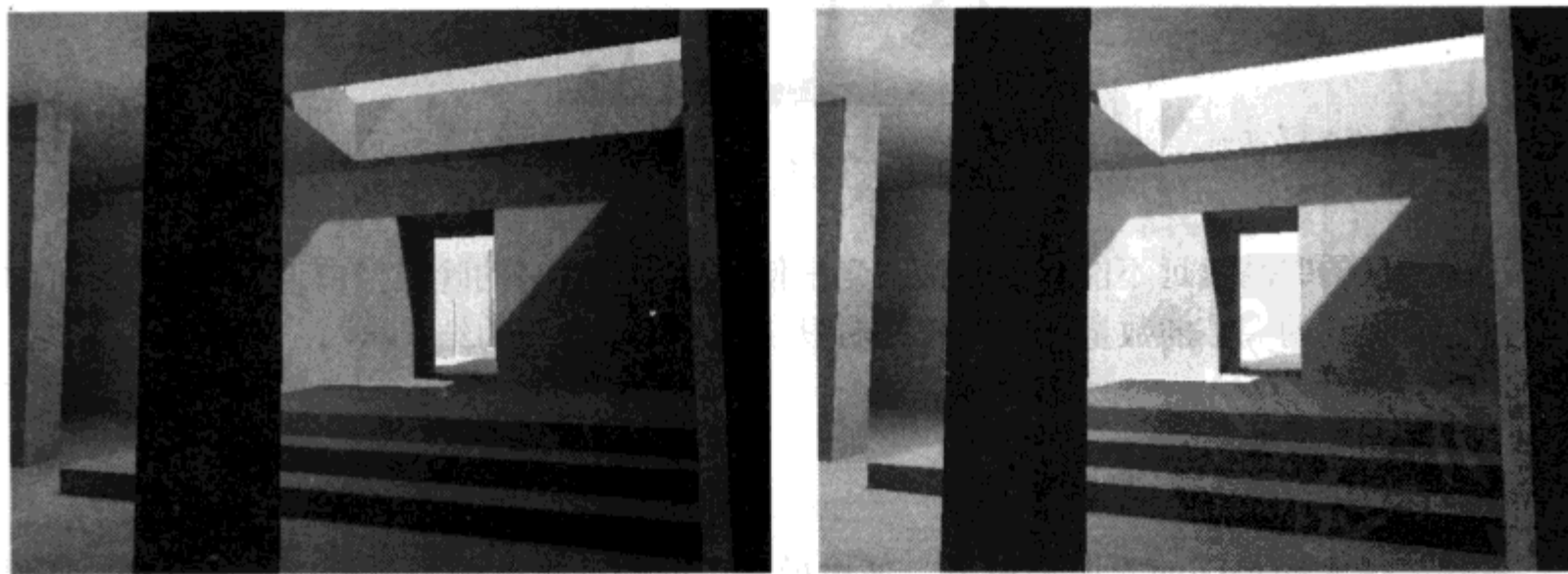


图 5.10.7 在黑暗的内部, 算法扩大了动态的范围, 有效地增量了图像

不只可以产生更逼真的图像, 它还可以提高游戏可玩性。例如, 当玩家从一个明亮的外边进入到黑暗的环境时, 他需要花费些许时间来观察周围的敌人。

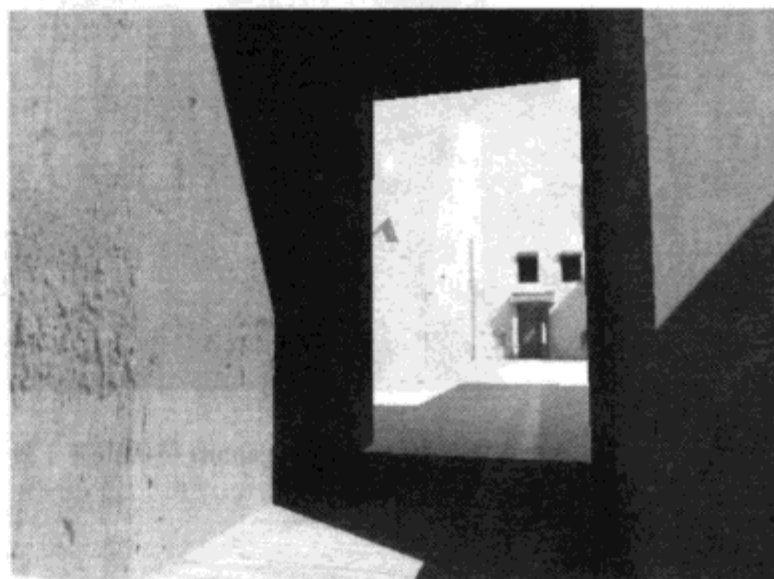
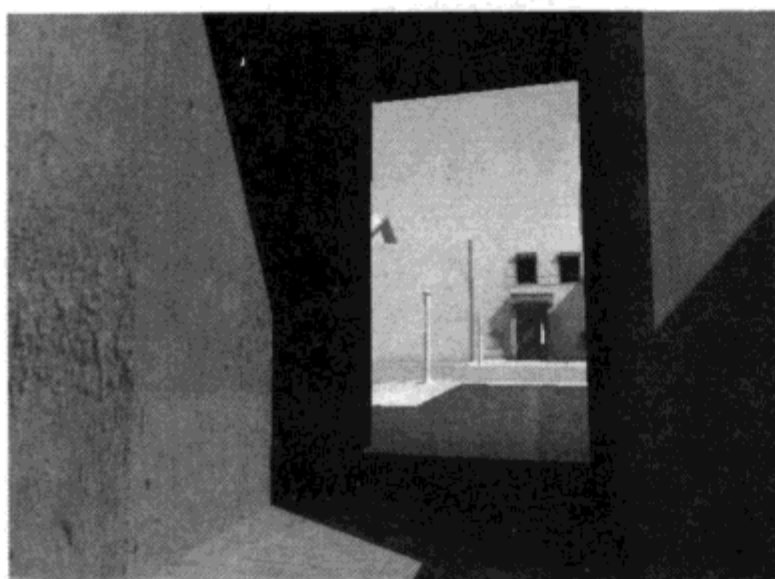


图 5.10.8 仍然是调整黑暗的内部，户外的太阳显得很灿烂

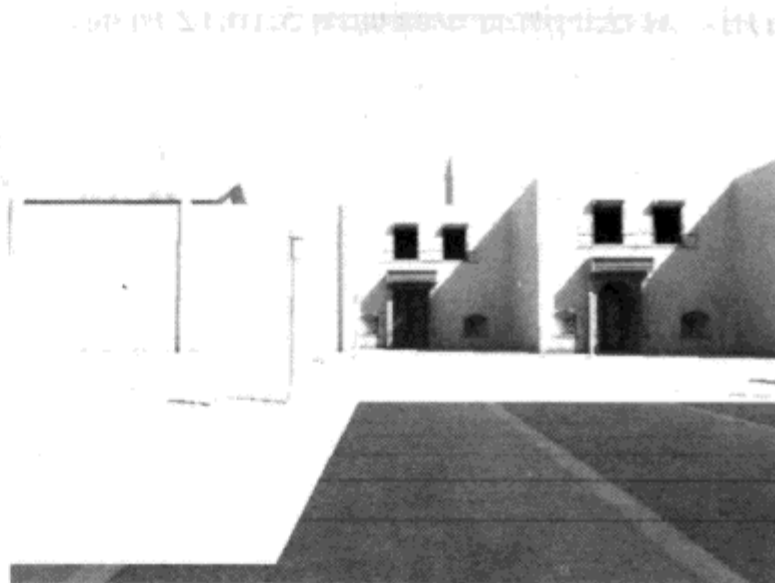
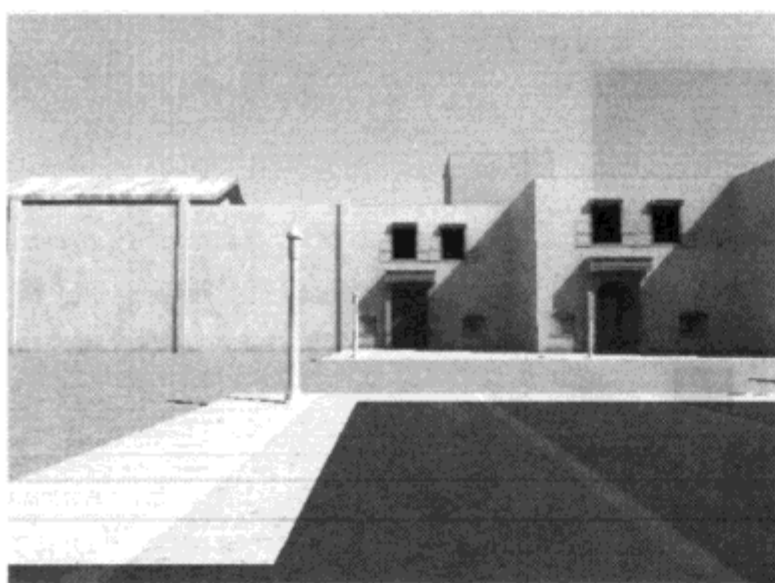


图 5.10.9 算法开始调整明亮的外部场景

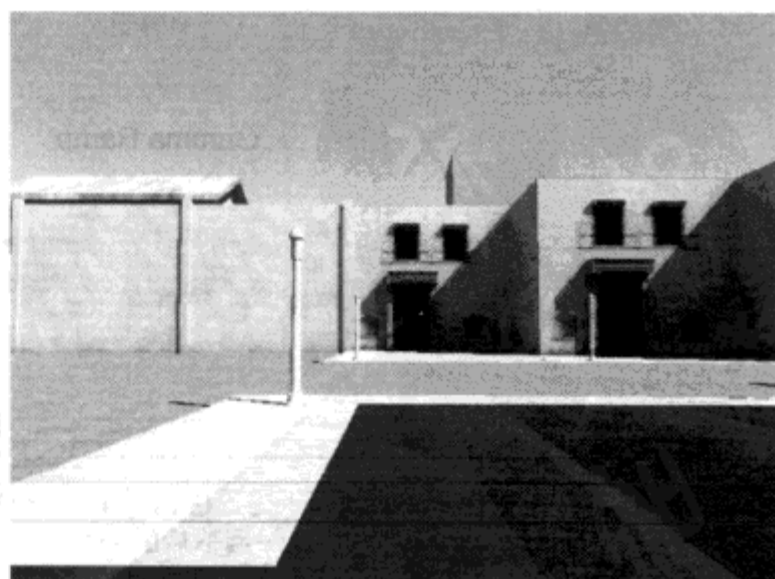
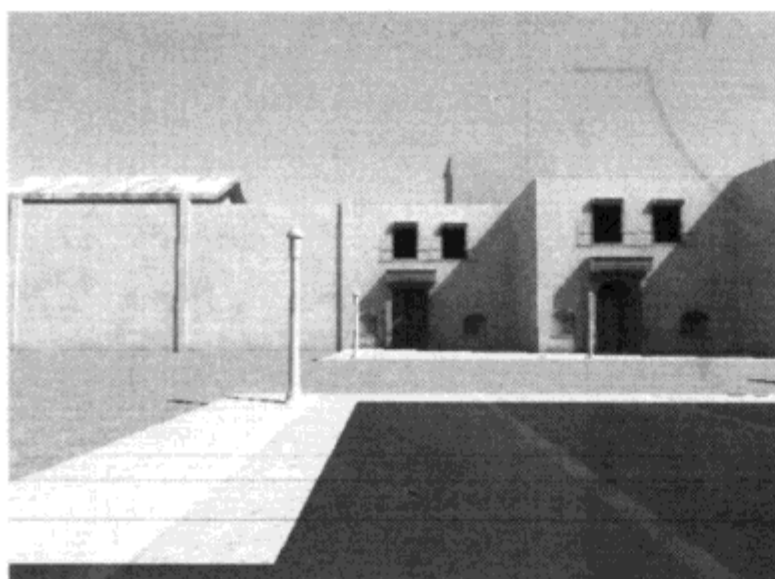


图 5.10.10 现在算法已经调整好了明亮的外部

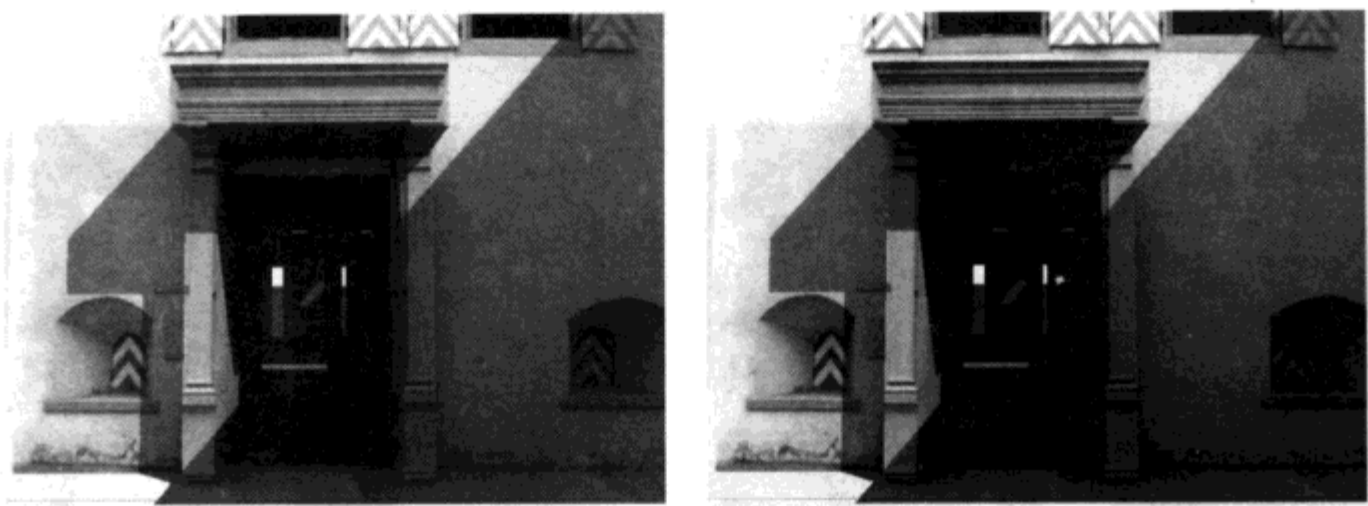


图 5.10.11 由于 gamma 等级偏向于外部明亮的等级，因此内部现在显得暗一些

5.10.6 算法

实现动态 gamma 的算法是十分简单的，它可以被分成不连续的几个步骤。CPU 和 GPU 的占用非常低。而且由于并非所有的步骤都需要完成每一帧，这进一步减少了对有效性能的占用。算法的所有步骤如图 5.10.12 所描述。

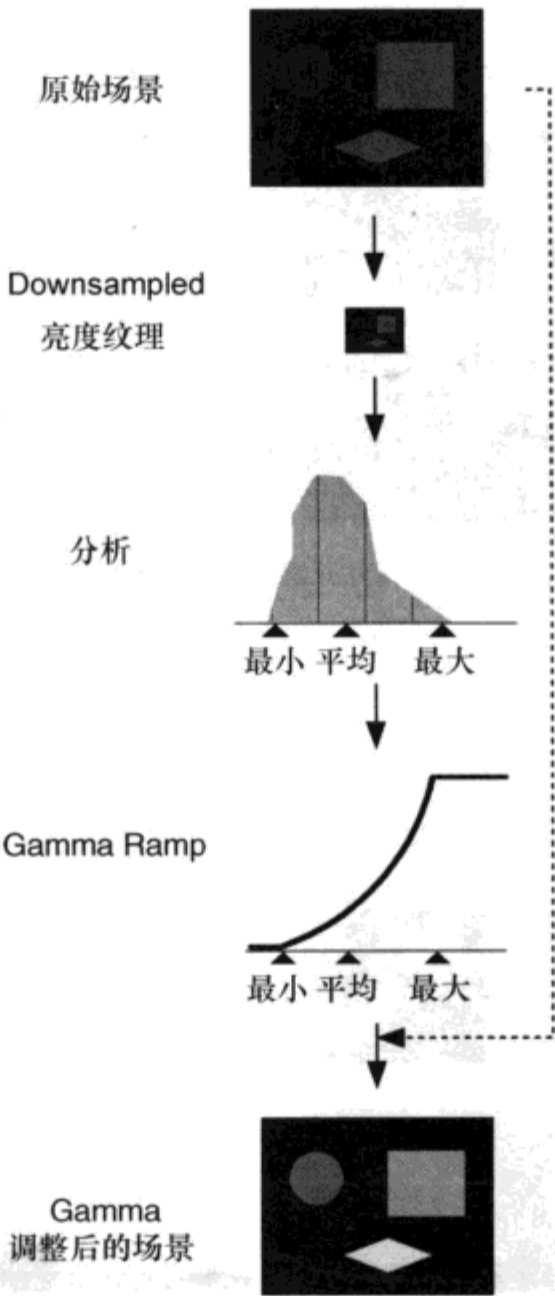


图 5.10.12 实现动态 Gamma 的步骤

1. 第一步：进行亮度转换时，Downsample 场景

首先，在进行亮度转换时使用 GPU downsampled 场景。最终渲染的图像作为纹理被使用，那些更小的每通道 8 位的纹理被设置为渲染目标。我们发现，把原始场景尺寸的十分之一作为目标尺寸就足够。例如，一个 640×480 的场景只需要一个 64×48 大小的 8 位亮度纹理。而且，由于我们对最小的和最大的场景亮度感兴趣，因此我们发现采样点在双线性的过滤上是首选的，这种过滤可以避免最暗的像素的明亮假象和最亮的像素的黑暗假象。亮度转换在像素阴影中完成。

$$\text{Luminance} = 0.30 * R + 0.59 * G + 0.11 * B$$

2. 第二步：明亮纹理的分布分析

接下来，使用 CPU 分析明亮的纹理。在 PC 系统上，较小的纹理必须从 GPU 存储空间拷贝到 CPU 存储空间。在使用统一的存储架构的系统上，视频存储到系统存储的拷贝不是必需的。

亮度分布通过小纹理的步进增长和计算每亮度值像素的数量产生。对亮度分布的分析可以确定场景的瞬间最小，最大和平均亮度值。为了排除局外因素，下边界和上边界的百分比作为计算最小和最大的亮度值来使用。平均值则由第五十个百分点定义。

```
struct Histogram
{
    float Lum[255];    // Lum[i] =在第 i 个亮度值处的像素片段
    float Min;         // 最小的亮度值
    float Max;         // 最大的亮度值
    float Avg;         // 平均亮度值
};

struct LuminanceImage
{
    uint NumPixels;
    byte* pPixels;     // 每个像素是一个字节，因为它
                      // 只保存亮度数据
};

void BuildNormalizedHistogram( const LuminanceImage* pImg,
                              Histogram* pHG )
{
    // 使用百分比来移除外层
    const float LowerPercent = 0.05f;
    const float UpperPercent = 0.95f;

    // 第 50 个百分点
    const float AveragePercent = 0.50f;

    // 通过对每个亮度值的像素片段
```



```

// 求和来构造归一化的柱状图
memset( &pHG->Lum, 0, 255 * sizeof(float) );
float InvNumPixels = 1.0f/pImg->NumPixels;
for( uint i = 0; i < pImg->NumPixels; i++ )
    pHG->Lum[pImg->pPixels[i]] += InvNumPixels;

// 寻找柱状图最小值, 最大值和平均值
float Sum = 0.0f;
for( byte c = 0; c < 255; c++ )
{
    Sum += pHG->Lum[c];
    if( Sum <= LowerPercent )
        pHG->Min = c/255.0f;
    else if( Sum <= AveragePercent )
        pHG->Avg = c/255.0f;
    else if ( Sum <= UpperPercent )
        pHG->Max = c/255.0f;
    else
        break;
}
}

```

3. 更新当前的 Gamma Ramp

接着, 被用来构建 **gamma ramp** 的最小值, 最大值和平均值通过一段时间的调节来模拟眼睛对明亮和黑暗的反应。

```

struct GammaRamp
{
    byte Ramp[255];    // 当前的值
    float Min;         // 当前的最小值
    float Max;         // 当前的最大值
    float Avg;         // 当前的平均值
};

void UpdateGammaRamp( const Histogram* pHG,
                    float Dt,
                    GammaRamp* pGR )
{
    // 如果变化大于 delta, 那么当前值就被更新
    const float Delta = 0.01f;

    // 下面的值可以被动态地更新
    // 以影响它所花费的用来适应变化的
    // 场景亮度的时间
    const float DMinDt = 0.1f; // 时间上最小值的改变
    const float DMaxDt = 0.1f; // 时间上最大值的改变
    const float DAvDt = 0.1f; // 时间上平均值的改变

```



```
// 在时间上更新当前的最小、最大、平均值
float Change;
Change = pHG->Min - pGR->Min;
if( fabsf( Change ) > Delta )
    pGR->Min += sign(Change)*DMinDt*Dt;
Change = pHG->Avg - pGR->Avg;
if( fabsf( Change ) > Delta )
    pGR->Avg += sign(Change)*DAvgDt*Dt;
Change = pHG->Max - pGR->Max;
if( fabsf( Change ) > Delta )
    pGR->Max += sign(Change)*DMaxDt*Dt;

// clamp 最小、最大和平均值以防止过度补偿
// 而导致极度黑暗或极度明亮的场景

const float LargestMin = 0.25f;
const float SmallestMax = 0.50f;
clamp( &pGR->Min, 0.0f, LargestMin );
clamp( &pGR->Max, SmallestMax, 1.0f );
clamp( &pGR->Avg, pGR->Min, pGR->Max );

// 斜线被用来稍微地饱和可见色
// 的值以阻止可见色显现出不饱和态
const float BiasMax = 0.5f;
float Diff = ( pGR->Avg - pGR->Min ) /
             ( pGR->Max - pGR->Min );
float Bias = 1.0f + Diff * BiasMax;

// 使最小值和最大值离散
byte Min = (byte) roundf( pGR->Min * 255.0f );
byte Max = (byte) roundf( pGR->Max * 255.0f );

// 根据新的最小、最大值和斜线设置 gamma ramp
// 所有小于最小值的值都被设置为 0
// 所有介于最小值和最大值沿着一条取决于斜线值的曲线设置
// 所有大于最大值的值都被设置为 255
byte c = 0;
for( ; c < Min; c++ )
    pGR->Ramp[c] = 0;
for( ; c < Max; c++ )
    pGR->Ramp[c] = (byte) ( powf( (float)( c - Min ) /
                                ( Max - Min ), Bias ) * 255.0f );
for( ; c < 255; c++ )
    pGR->Ramp[c] = 255;
}
```

4. 在场景中应用当前的 Gamma Ramp



最后, gamma ramp 被用来更新最终的场景。这也可以通过使用像素阴影在硬件中被完成, 或者通过直接设置硬件 gamma ramp 来完成。任何一种方法都有其正面和反面。设置硬件 gamma ramp 不需要额外的全屏传递场景就可以应用 gamma ramp, 但是那些不应该被场景光影响的元素(比如 HUD 和一些覆盖图形)将需要使用一个可应用在 pixel shader 中的相反的 gamma ramp 来调整(在随带的 CD-ROM 上的代码中有详细的说明)。使用一个 pixel shader 是非常昂贵的, 因此图像可以在 HUD 元素被绘制之前先在后台缓冲中被绘制, 并且可以使用其它的后处理技术比如 blooming 和景深加入到传递中。

5.10.7 结论



最新的硬件支持更高精度的渲染目标格式。使用这些更高精度的格式, 这种技术越来越让人感到兴奋, 因此场景可以在减少波形的情况下得到很好的调节, 并且一些超过被选择的亮度最小值的颜色值可以被找到。由于这是一种后处理技术, 因此它应该很容易地被整合到目前已存在的引擎中。随书附带光盘中的这项技术的所有源代码都是可用的。



5.11 热和薄雾的后处理效果

作者: Chris Oat and Natalya Tatarchuk, ATI Research, Inc.

E-mail: Coat@ati.com, Natasha@ati.com

译者: 刘永静

审校: 谷 超

同几年前的加速器相比,当前的硬件图形加速器允许程序员有更高的像素填充速度。然而,由于游戏程序员为了使自己的产品努力符合最主要的“最低标准”市场的要求,因此这些带宽并未被使用。添加后处理(post-processing)特效是一种既可以充分利用图形硬件的强大性能又不会破坏你的游戏渲染管道的简单方式。这些效果可以很容易地被添加到几乎是任何的渲染管道的后期制作,程序员只要稍微做些相关工作,就可以创建出极好的视觉效果,通过这些效果能够极大地提高最终的输出效果。

5.11.1 热和闪光的薄雾

在炎热的夏天,在户外的人或许都可以观察到一个热表面上热气发出微光(heat shimmering)的视觉效果,比如沿着高速公路铺设的被太阳烘烤过的沥青。这种 shimmering 在技术上被称为下蜃景(inferior mirage[Berger90]),是由于接近地面的空气温度高于上空所造成。因为在沥青表面上空很远的地方冷空气的密度比离地面近的地方高,靠近沥青的上方的空气热膨胀时光线穿过造成了这样的结果。因为随着密度的变化气体表现出不同折射指数(indices of refraction),所以当光线穿过不同温度的气体时折射不同。一个热源,例如熔岩池,拥有大量的,快速移动的气流对流(convection currents),将展现出一个显著的和生动的热闪光特效。

5.11.2 高级算法

一个高度精确的热闪光(heat shimmering)物理模拟对大多数游戏引擎来说杀伤力过大,因此是没有必要的。本文取而代之的是,集中于通过近似的一幅图像再现气体对流(gaseous convection)的视觉现象,因此看起来赏心悦目。图 5.11.1 显示了 ATI Caves 演示的屏幕截图,其图解说明了在这篇文章中描述的热薄雾(heat haze effect)特效。基本的算法如下:

- (1) 创建一个可渲染的 (renderable) RGBA 纹理其大小同后缓存 (back buffer) 一样。
- (2) 清空可渲染的纹理为 (0.0, 0.0, 0.0, 1.0)。
- (3) 渲染整个场景到上面提到的可渲染纹理, 写入颜色到 RGB 和深度/变形标量到 alpha 通道。
- (4) 绑定一个可渲染纹理成为纹理单元的其中之一并渲染一个 screen-aligned 四边形到后置缓冲 (back buffer)。用归一化设备坐标 (NDC) 作为纹理的纹理坐标, 来读取存储在 alpha 通道中的失真标量 (distortion scalar)。通过失真标量偏移 NDC 坐标, 并重新取样纹理来得到失真 RGB 值, 然后输出这些到后置缓冲 (back buffer)。

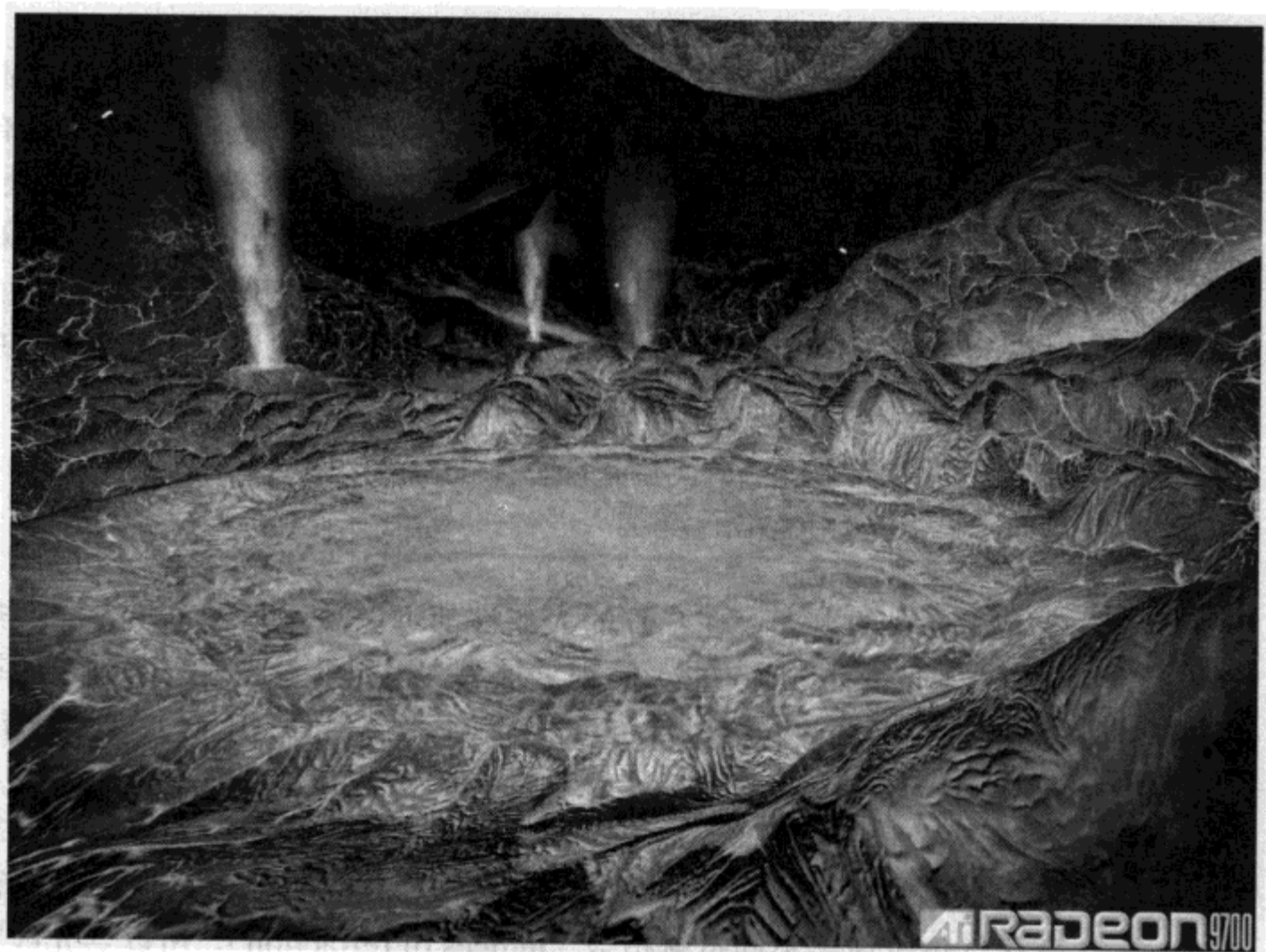


图 5.11.1 洞穴演示程序中的热薄雾效果

5.11.3 计算失真值

逐像素 (per-pixel) 失真值是必需的, 它被用来确定在给定的像素上应该应用多少的失真。在这里介绍三种不同的方法产生失真值。第一种方法是用逐像素场景 (per-pixel scene) 深度值来确定失真权重。这种方法实现起来简单, 并能产生相对较好的结果。更复杂的实现受益于热的几何图形 (heat geometry) 和热纹理 (heat textures) 在和场景深度联合中的使用。虽然这些方法中的任何一个实现都能用来生成失真值, 但是这三个方法联合较为理想。

1. 场景深度

当光线穿过大气层, 它会遇到多种不同密度的气袋 (pockets of gases)。光线交叉的气

袋越多，光线折射的可能性就越高。因此，场景深度在模拟热薄雾效果（heat haze effect）中起到很重要的作用。通过投射顶点到视觉空间并提取该顶点位置的 z 分量（z-component），很容易计算一个逐像素成分（a per-vertex basis）到眼睛的距离。然后这些逐像素深度值被传到像素着色器（pixel shader）。使用一个代表最大失真的 alpha 值首先清空目标缓冲（destination buffer）是很重要的，这样没有被写过的帧中的任何像素将被看作是在无穷远处（只有当你不绘制帧缓冲中的任一个像素时，这才是必需的）。通过写这个深度值到目标 alpha，并不能够完成一个简单的实现；然而，这种过分简单化的方法只是为整幅图像提供恒定失真。

2. 热的几何图形

为了渲染一个失真值到一个屏幕外缓冲的 alpha 通道，一些几何图形必须被绘制。例如，一个热熔岩池可能有一些穹顶状的网格（domelike mesh）在它上面。图 5.11.2 显示了熔岩池中心的穹顶形网格线框。这种网格担当我们的“热的几何图形（heat geometry）”，因为它的惟一目的就是绘制热的失真值到 alpha 通道。

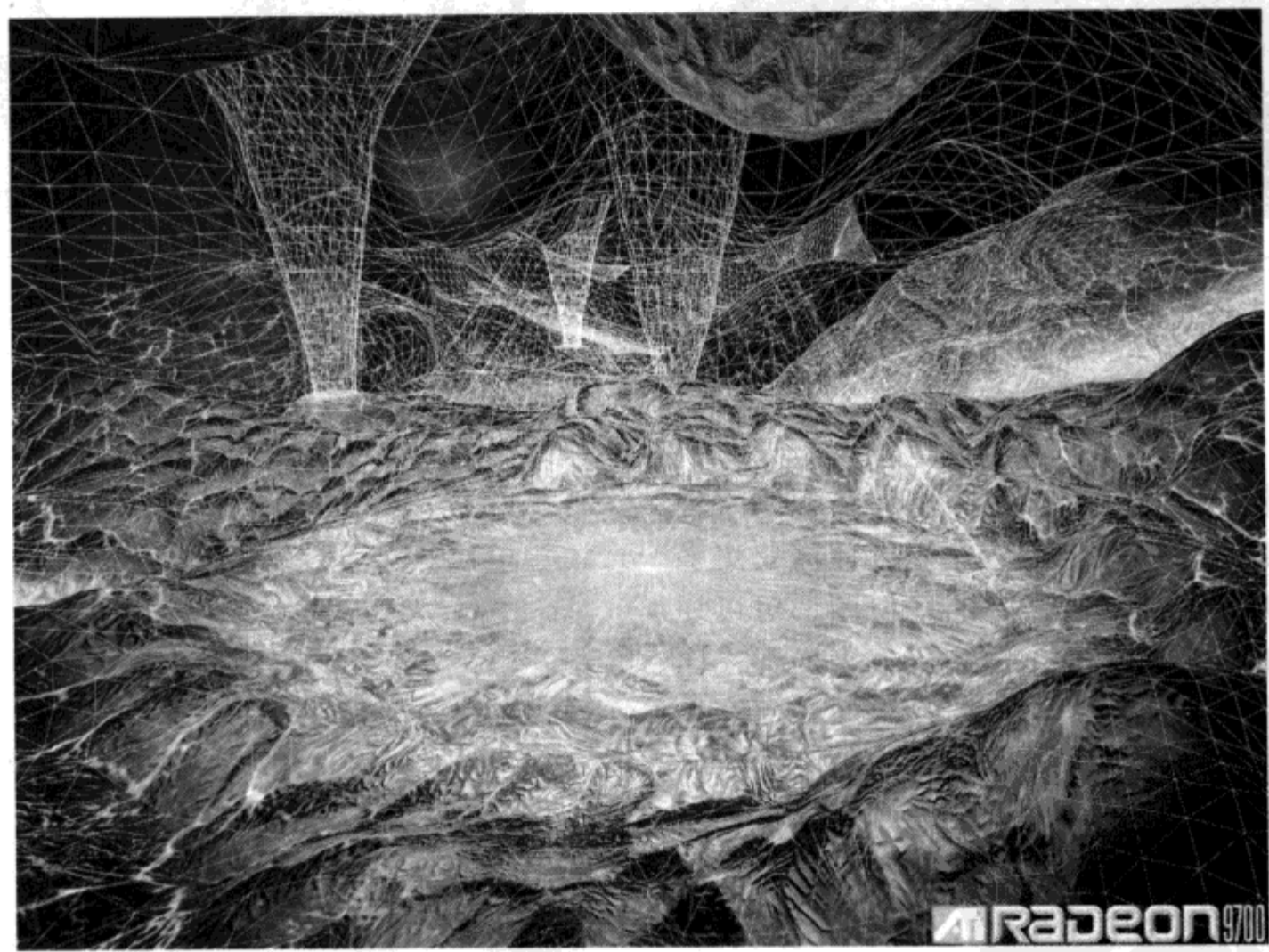


图 5.11.2 热熔岩池的失真几何图形网络框架

当绘制热的几何图形时，要特别注意避免绘制陡边（sharp edge）到屏幕外缓冲的 alpha 通道。因为被模拟的热的几何图形包围一体积气体，硬边线创建了令人烦扰的可见的遗留物（disturbing artifact），因为没有被包含的气体充满它们的环境而且并不显示任何可见边。如果我们看看热沥青表面上面热闪光的例子，闪光的开始和结束没有明显的线。因此，为了维持

这种效果的视觉完整性，我们需要这避免些硬边；由热的几何图形生成的 α 值应该朝向轮廓边线慢慢渐变。对穹顶 (dome) 和孔口 (vent) 来说，通过 $N \cdot V$ 放缩 α 很容易就能够达到，这样 α 渐变表现为平面法线偏离视觉向量。图 5.11.3 显示了一个热孔用热的几何图形 (heat vent) 模仿正在发热的气体。

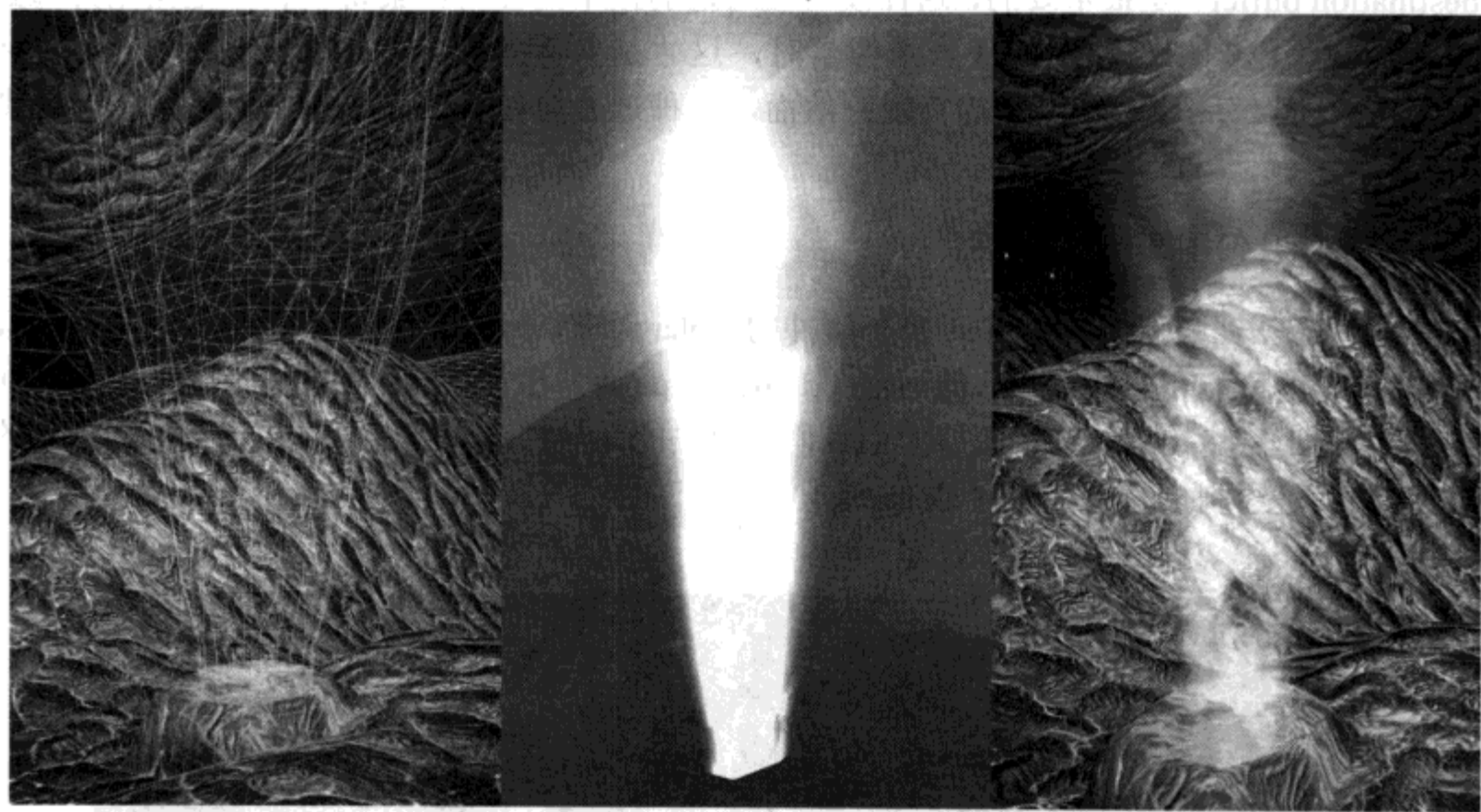


图 5.11.3 (左图) 用丝网框架模型绘制的热气出口的热几何图形。(中间) 为了消除硬边，经过高度和 $N \cdot V$ 的比例缩放处理后的出口的失真值。(右图) 最终的失真图

3. 热纹理

热的几何图形在包围均匀体积的气体方面工作得很好。当一体积气体需要更多的随机密度分布时，热的几何图形可以通过应用热纹理 (heat texture) 来增强，比如图 5.11.4 所示。热纹理在热的几何图形的表面滚动穿过，通过本身的深度和热的几何图形所提供的失真值调制每个像素。这些纹理滚动的方向是很重要的，它们依赖于将要被应用的热的几何图形类型。从热的公路上升起的热气在世界空间中应该朝上滚动以便来模拟气流对流 (convection currents)，而喷气发动机的热流应该朝外，同发动机的通风口一致。图 5.11.5 显示了组合的场景深度，热的几何图形和热纹理。

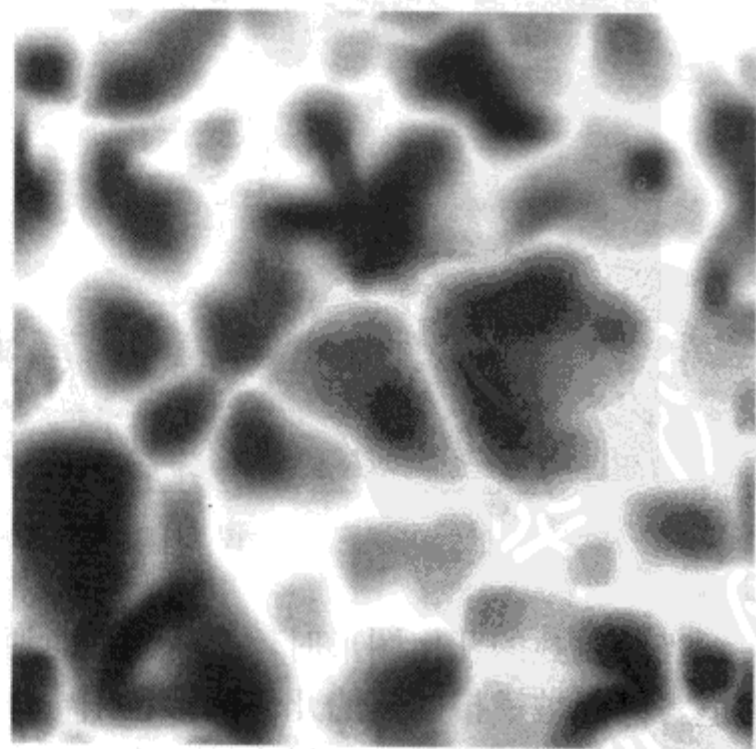


图 5.11.4 一个在熔岩中使用的热纹理的例子

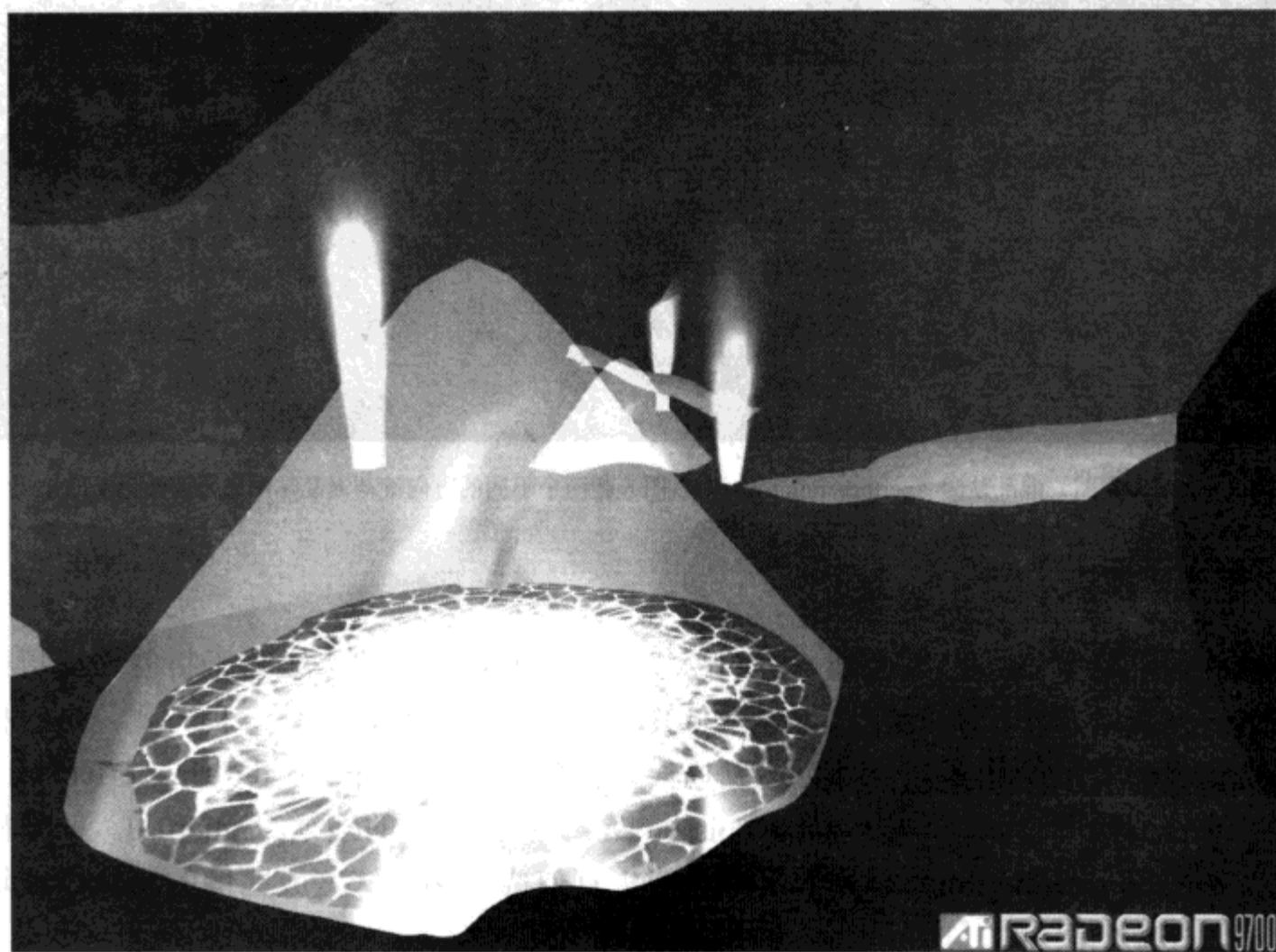


图 5.11.5 所有对最终失真值有贡献的场景深度，热几何图形，热纹理

5.11.4 失真值的解释

到目前为止，我们已经介绍了不同的将失真值渲染到屏幕外缓冲的 alpha 通道的方法。下一步是绑定屏幕外缓冲为一个纹理，然后绘制全屏（full-screen）四边形到后台缓冲（back-buffer），其解释了在 alpha 通道中的逐像素失真值并因此修改了 RGB 通道。

1. 逐像素混乱

一个混乱贴图（perturbation map）可以用来模拟光线穿过不同密度的大气层时所折射的弯曲的路线。一个混乱贴图是一个包含逐像素纹理偏移的纹理。这同被凹凸贴图（bump mapping）使用的标准图是相似的，由于我们在图像空间里使之混乱，因此这些标准图只需要是 2D 的就行了。这个单独图能够有效地在 vertex shader 在两个不同方向被滚动，同时在 pixel shader 里需要被逐像素抽样两次，从而得到两个混乱向量（perturbation vector）（见图 5.11.6）。通过存储在我们外屏幕（off-screen）渲染目标的 alpha 通道的失真值，这些向量被平均分布及缩放。这些被缩放的向量用规一化的设备坐标总计，并被一个同提取外屏幕渲染目标相关的纹理使用以恢复一个已被混乱的 RGB 场景像素。


```

        float2(0.473434f, -0.480026f),
        float2(0.519456f, 0.767022f),
        float2(0.185461f, -0.893124f),
        float2(0.507431f, 0.064425f),
        float2(0.89642f, 0.412458f),
        float2(-0.32194f, -0.932615f),
        float2(-0.791559f, -0.59771f));

    // 中心 tap
    cOut = tex2D (tSource, texCoord);

    for (int tap = 0; tap < 12; tap++)
    {
        float2 offset = (pixelSize*poissonDisc[tap]*discRadius);
        float2 coord = texCoord.xy + offset;

        // 采样像素
        cOut += tex2D (tSource, coord);
    }

    // 平均并返回
    return (cOut / 13.0f);
}

```

这种逐像素模糊的形式对一些以前的图形加速器来说代价太高了。有一种方法花费的代价较少,但可用来达到相同的效果,那就是对外屏幕缓冲的 **down-sampled** 拷贝执行可分离的高斯模糊 (**Gaussian blur**), 然后根据失真值在模糊和未模糊的缓冲之间进行线性插入。

下面是这种特效的完全的 **HLSL pixel shader**。

```

sampler tRBFULLRes; // 外屏幕缓冲 (用 alpha 失真)
sampler tNormalMap; // 混乱贴图

float fBumpStrength;

struct PsInput
{
    float2 texCoord0 : TEXCOORD0; // 静态的屏幕正方形坐标
    float2 texCoord1 : TEXCOORD1; // 卷轴坐标
    float2 texCoord2 : TEXCOORD2; // 更多的卷轴坐标
};

float4 main (PsInput i) : COLOR
{
    // 从带有卷轴坐标的混乱贴图中获取
    float3 vNormal0 = tex2D (tNormalMap, i.texCoord1);
    float3 vNormal1 = tex2D (tNormalMap, i.texCoord2);

    // 测量和斜线
    vNormal0 = SiConvertColorToVector(vNormal0);
    vNormal1 = SiConvertColorToVector(vNormal1);
}

```



```
// 求和和测量
float2 offset = (vNormal0.xy + vNormal1.xy) * fBumpStrength;

// 从渲染的纹理中得到颜色和失真信息
float4 cScene = tex2D (tRBFULLRes, i.texCoord0);

// 将失真值平方
cScene.a *= cScene.a;

// 计算混乱纹理的坐标
offset.xy = ((offset.xy * cScene.a) * fBumpScale);
float2 newCoord = i.texCoord0 + offset.xy;

// 假定 1600×1200 大小的外屏幕缓冲的像素尺寸
float2 pixelSize = float2(1.0f/1600.0f, 1.0f/1200.0f);

// 取得失真的颜色
float4 o;

// 最大的圆盘半径是 5.0 个像素
o.rgb = SiGrowablePoissonDisc13FilterRGB(tRBFULLRes,
    newCoord, pixelSize, 5.0f * cScene.a * cScene.a);

o.a = 1.0f;

return o;
}
```

5.11.5 结论

本文示范了一种作为后处理步骤的方法，那就是添加热和闪光的薄雾。这种效果作为一系列衡量图形复杂度的步骤被提出来；以前的图形加速器可以完成简单的混乱扭曲，而性能更高的硬件可以添加额外的热的几何图形，热纹理，和逐像素模糊。本文的更进一步扩展之一是加入粒子系统（particle system），用来添加更复杂的气体/物体交互。特别感谢 Eli Turner 提供本文所用到的图像美术。

5.11.6 参考文献

[Berger90] Berger, M., T. Trout, and N. Levit, *IEEE Computer Graphics and Applications*, pp. 36–41, May 1990, Vol. 10, Issue 3.

[Riguer03] Riguer, G., N. Tatarchuk, and J. Isidoro, “Real-Time Depth of Field Simulation,” *ShaderX2: Shader Programming Tips & Tricks with DirectX 9*, Wordware Publishing, Inc., 2003.

5.12 用四元数的硬件蒙皮

作者: Jim Hejl, Electronic Arts Tiburon

E-mail: jhejl@ea.com

译者: 刘永静

审校: 谷超

给 定一个已有细节的皮肤网格 (skin mesh), 和一个相关的动画层级, 我们的目标是当底层骨架动起来的时候, 产生一个自动地变形网格的方法, 这个过程被称为蒙皮 (skinning)。

现在, 大多数视频游戏角色蒙皮是用一种线性混合技术完成的, 称为顶点混合 (vertex blending) (也就是在 Maya 里通常所说的骨架—子空间变形 (skeletal-subspace deformation), 矩阵调色板蒙皮 (matrix palette skinning), 和平滑蒙皮 (smooth skinning)。虽然流行, 但顶点混合 (vertex blending) 也由于其缺点而声名狼藉。最严重的人工效应物, 线性近似的一个直接后果就是, 随着偏转角的增加会引起关节物理上的崩溃 (physically collapse)。作为一个戏剧性的论证, 简单的扭转关节会产生特有的糖纸 (candy wrapper) 变形。图 5.12.1 图解说明了这种效果, 它显示了一个关节同时扭绞和旋转。

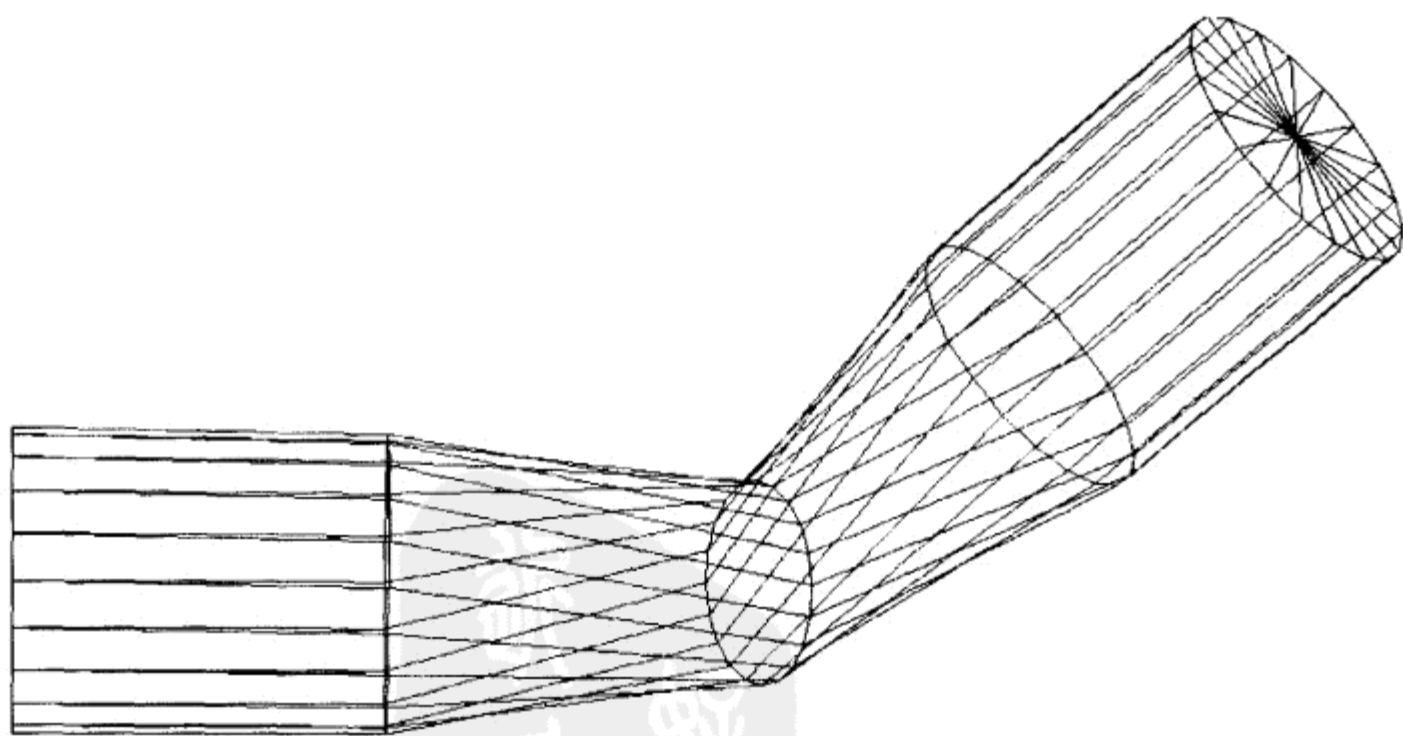


图 5.12.1 线性的顶点混合

带来的副作用是, 通常必须忍受相对简单和基本算法的效率作交换。

实际上，当然整个解决方案使用用户级可编程顶点处理（programmable vertex processing）很容易在硬件里实现。

顶点混合效果很差，因为它缺乏球形插值（spherical interpolation）的概念。这篇文章介绍了球形关节混合（spherical joint blending），一种交替蒙皮算法（alternative skinning algorithm），其没有线性副作用。球形关节混合是用四元数实现的，并且完全在一个 vertex shader 中执行。正如我们将要看到的，最终的实现比起顶点混合的方案，用了较少的指令以及更少的 GPU 内存。图 5.12.2 显示了用相同的偏转角但是改进的变形。

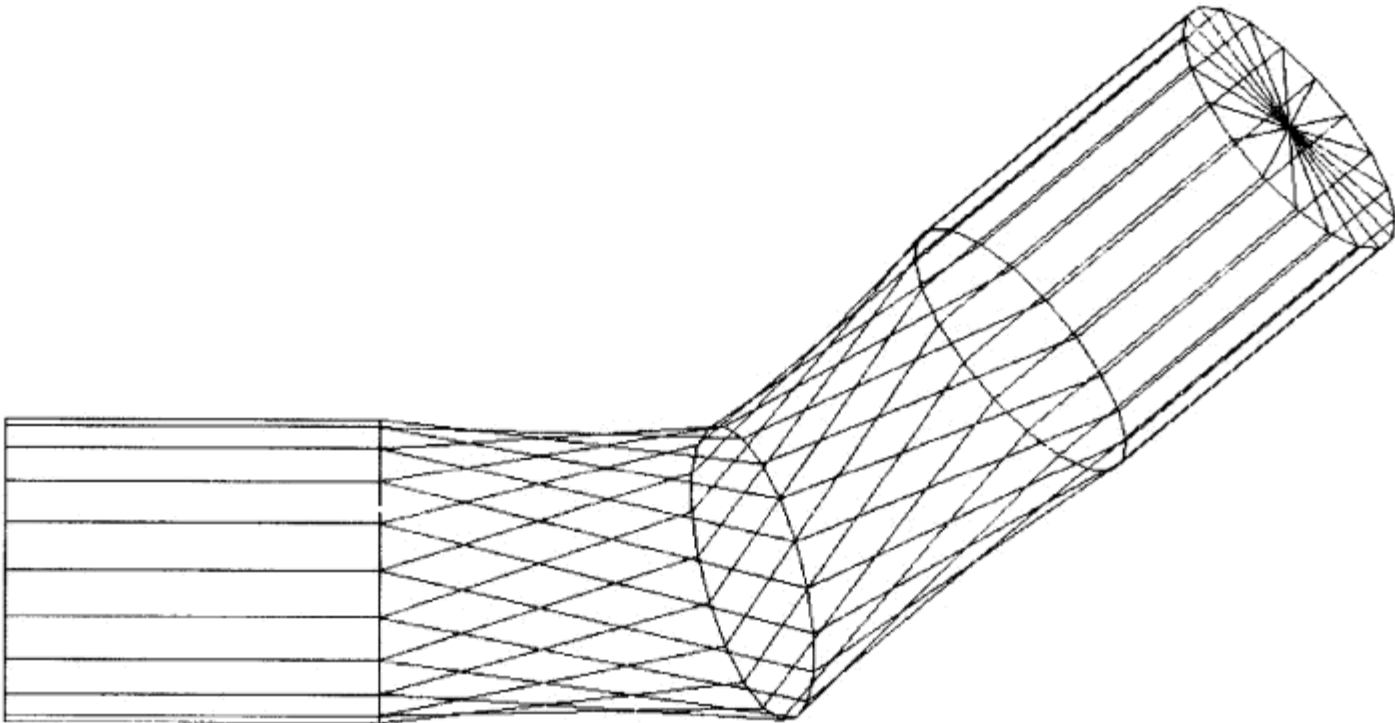


图 5.12.2 球形的关节混合

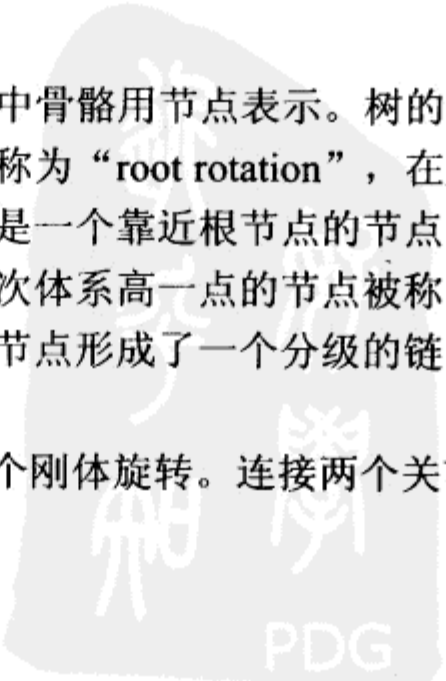
5.12.1 蒙皮的概念

基本的蒙皮算法首先通过放置一个分层的骨架到一个角色静态的模型中，然后把两者绑定。骨架和网格是要小心对准的，尤其是一些中立姿势。然后，每个在网格中的顶点被赋值为一组有感应的关节和为每一感应的混合的权值。因为潜在的骨架（underlying skeleton）是动的，所以移动骨架网格就变形了。这种变形，移动头顶骨骼的感应，就是 skinning 算法的任务了。

1. 骨架概述

骨架可以被看作是一种树形结构，其中骨骼用节点表示。树的最高节点是根节点，其对应分层的根对象。根节点的变换和旋转被称为“root rotation”，在层次体系中所有其他节点的变换都与它相关。在层次体系中高节点是一个靠近根节点的节点。而后继节点是远离根节点的节点。当比较同其他相连节点，在层次体系高一点的节点被称为父节点，而后继节点被称为子节点。从 skeleton 的根节点的后继节点形成了一个分级的链，子节点的坐标系总是同父节点的坐标系相关联。

骨架中的节点是关节（joint），是一个刚体旋转。连接两个关节的段是一个骨骼位移，



也是一个刚形体变换。一个关节旋转的完整变换接着骨骼位移就是骨骼。

2. 顶点混合概述

顶点混合算法变换一个顶点多次，一次为每个骨头感应，并且用对应的存储在顶点混合加权值来混合结果在一起。为了使这种内插工作，在变换以前定位顶点到每一目标骨头是必需的。详细资料见[Domine03]。这种定位在一个专一坐标系中有 skinning 顶点多次的效果。这些结果是被加权的及被累积到一个最终的顶点。

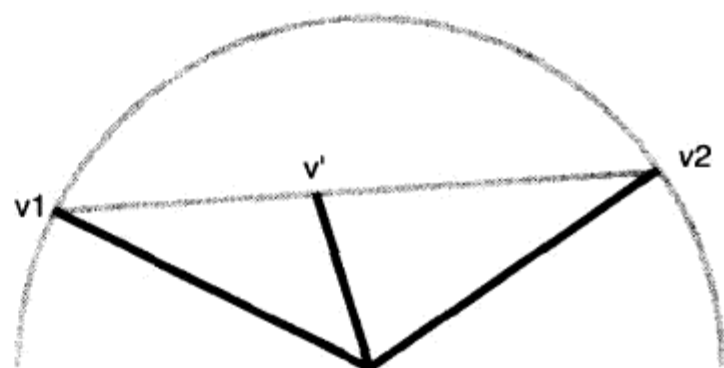


图 5.12.3 顶点混合

注意到用 v_1 和 v_2 之间的直线来创建子空间。顶点混合的第一个主要缺点直接由这样的事实引起的，就是 v 的变形被限制在这个子空间里。这个子空间随着偏转角的增加会继续崩溃，最后在 180 度陷入奇点。在这个子空间的变形不是想得到的；因此无论怎样调整混合的加权都不会产生想要的结果。

线性的顶点混合与直接在骨骼矩阵之间内插在本质上是相同的。也就是说，矩阵变换点的加权和等于变换点加的加权[Shoemake92]。理解这一点非常有用，因为大多数的图形工程师对企图内插矩阵的后果很熟悉。我们能够想象以内插值替换矩阵的基向量变得非正常了，如在图 5.12.3 中的向量。加之，以内插值替换的基向量可能不是互相垂直的，在变换中导入了歪斜。直接的矩阵内插其辩论科学增加了我们对顶点混合为什么会失败的理解。

怎样在多个加权旋转之间内插不是直接就很明显的。事实上，三维空间的旋转不是一个简单的向量空间，而是一个闭合的三维簇。如果你对术语不熟那不用担心；这个概念很简单。一个闭合簇仅仅是这样的东西，其局部好像欧几里得空间（平面），但实际上背靠自己弯曲。这就是为什么地球看起来似乎是平面以及为什么顶点混合看来似乎为小角工作。这种三维簇的旋转空间的记法为 $SO(3)$ ，用于特殊正交（特殊正交是一组“真旋度”，正交矩阵的行列式（determinant）1）。

正如所提到的，因为 $SO(3)$ 的拓扑结构，顶点混合看起来似乎是为小角工作。先前的工作成果在《游戏编程精粹 3》[Weber02]以及 SIGGRAPH 2003 [Mohr03]中有先前的工作成果，它们建议添加额外的关节到骨架来分解大偏转角。描绘一个分成多个小角的旋转，如图 5.12.4。在这些角之间线性内插将创建分段线性逼近，其开始会聚于我们实际想要的：球形插值。

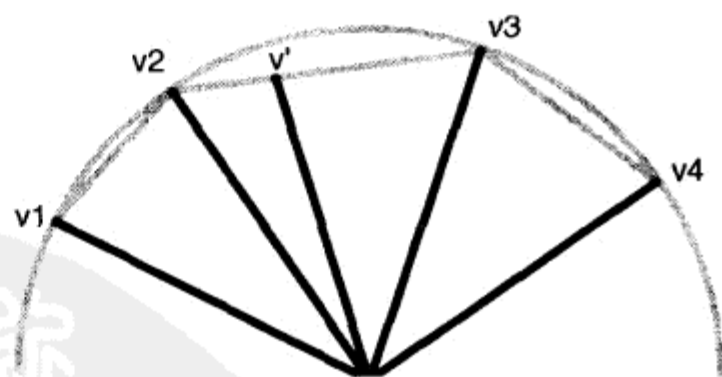


图 5.12.4 分段线性近似值

3. 关节混合介绍

与其求变换顶点的平均数，不如假定我们移动插值下至关节。更确切地说，与其认为是合成一个蒙皮的向量，不如认为是合成可以正确地蒙皮向量的变换。

在 mesh 中的每个向量被严格地限制在一个惟一的骨头，但是关节的旋转是几个关节的平均数。换句话说，我们通过求几个其他坐标系的平均数综合了一个新的坐标系。新坐标系的位置是通过分层的附属骨头的位置而给出的。（因为骨骼在动画期间作为特色不会改变长度，所以它们被设想为沿着关节链一连串的常量位移）。合成的坐标系 M' ，受多个关节影响而是旋转的。如图 5.12.5 所示，坐标系 M' 会变换顶点 v 直接到蒙皮位置 v' 。

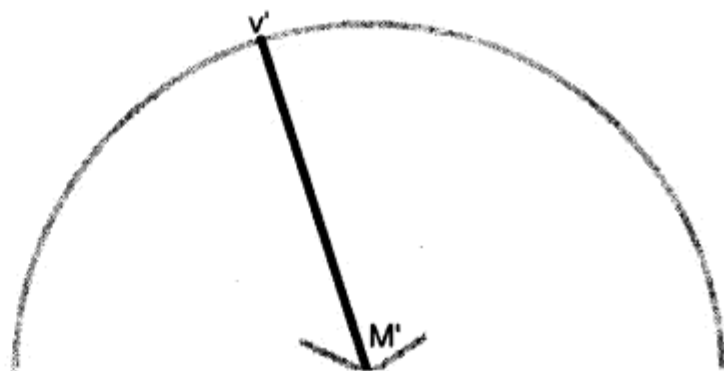


图 5.12.5 球形的关节混合

那么， M' 是怎么合成的？我们已经看到了，在旋转矩阵之间的内插没有起作用。事实上，具有矩阵内插的关节混合会产生顶点混合的结果。一个新的旋转参数化是被需要的。

5.12.2 四元数参数化

回忆一下，四元数是 4 个合成的数构成的四维向量空间。单位四元数的组合形成了一个四维球形，就是通常所说的 S^3 。如果你不熟悉这种记法，“3”代表拓扑结构，不是几何学的，而是定义。几何（geometers）指在基础空间坐标的数量，而拓扑（topologists）指表面自己的维数[Coxeter73]。想象 S^3 是两个固态球体粘结在一起。只一个普通球体是两个圆盘沿着它们的边界环绕胶合在一起成为联合，两个固态球形体接在一起创建一个超球体。

特别有趣的是， S^3 碰巧是 $SO(3)$ 的两倍。更确切地说，单元四元数是两倍于三维旋转群。从技术上就是通常所说的通用覆盖（universal covering），这就允许单元四元数以最简单的可能形式捕捉三维旋转的所有几何和拓扑结构[Shoemake94]。这种相互关系的细节是令人神往的，但远超出了这篇文章的范围。

1. 球形体内插值

正如我们所见，单位四元数用 S^3 球体上一个向量表示旋转。这样，四元数线性化旋转群，并没有逼近它。内插一个旋转现在直接了当就是内插一个惟一向量。

在两个旋转之间内插应该给出在 S^3 上的两个点之间的圆弧上，给出点。这种穿过超球面表面的内插被称为是球状线性插值（spherical linear interpolation），或者 SLERP。给定四元数 p 和 q 其为在它们之间的锐角，SLERP 在位置 w 被定义[Shoemake85]。

$$\text{slerp}(w;p,q) = \frac{\sin((1-w)\theta)p + \sin(w\theta)q}{\sin(\theta)} \quad (5.12.1)$$

作为性能优化，我们可以逼近 SLERP 为

$$\text{slerp}(w;p,q) = \|(1-w)p + wq\| \quad (5.12.2)$$

这种逼近会产生定位，其被视为在四元数空间里一条直线上的四维点。这种以内插值替

换的四元数在内插期间不会维持单位长度,所以结果被投射回超球面。由于 SLERP 廉价逼近描绘出正确的相同曲线,但是这种穿过球形体的快速操作结果是以非恒定的速度穿过圆弧。我们要接受这种为更简单的数学上的权衡。

在合成一个以内插值替换的四元数之后,在变换向量之前需要转换矩阵的形式。幸运的是这种转换是一个简单的多项式链,不需要额外的修饰。因此,内插一个四元数并且转换它到一个旋转矩阵,能够用简单的算术指令在向量硬件上实现。

2. 对映的四元数

我们早先提及 S^3 是 $SO(3)$ 的两倍。让我们深入看看。假定我们有一球形体上的两个点 A 和 B 。有一个短弧连接 A 到 B (最短程线),而一个长弧连接 B 到 A 。因为每一旋转有两类路径,通用的覆盖群 S^3 有 $SO(3)$ 两倍多的元素。这意味着两个不同的单位四元数能被表示为在 $SO(3)$ 里同样的旋转。这可能有助于把 $SO(3)$ 视为 S^3 的投影,每对在 S^3 中的对映点投射成 $SO(3)$ 中惟一的点。

这样的双倍覆盖有插值法结果。当在两个单位四元数之间内插,我们总是对球形体上的短程线感兴趣,而不是环绕的“长距离”。为了实现这些,我们挑选那些两者之间是非负的点积。这样具有选择短路径的效果。对于蒙皮,这意味着所有骨架的四元数必须被强制在同样的(4D)半球。这个算法简单:测试相对于第一个关节的在骨架中的每一关节。如果两者之间的点积是负的,在选择关节处倒置此四元数。这将保证所有骨架点的旋转,到这样的半球,它是通过由正确的插值得来的第一个关节定位而定义的。这样的操作应该发生在 CPU 装载四元数到常量存储器之前。跳过这个步骤将会产生无法预料的结果。

5.12.3 硬件实现

为硬件蒙皮做准备,CPU 加载关节变换到 vertex shader 的常量寄存器:一个四元数旋转和在骨架中每一关节的骨骼平移。这用了两个四边形寄存器的 7 种元素。(没有用到的第 8 种元素能被用来存储均匀标尺,因为四元数不能包含一个标尺。)这样,一个有 28 块骨骼的角色可以用 56 个 vertex shader 寄存器。

在微代码中,GPU 会在多个关节旋转中用相应的存储在每个顶点的混合加权,来内插。混合的四元数将被转换为一个 3×3 的旋转矩阵。旋转矩阵会被用来蒙皮必须的向量(位置、法线、副法线、切线,等等)。这个顶点被假定严格的限制在首次的骨骼感应,并且通过它的骨骼位移被平移。

1. 球形体内插近似值

在 Microsoft 的 DirectX 顶点汇编着色语言(vertex assembly shader language)中,每个四元数是加权和混合的。

; 四骨骼内插值

```
mov    a0.x, v[BONE].x
```

```

mul    r0, c[a0.x], v[WGT].x    ; 第1个四元数
mov     a0.x, v[BONE].y
mad     r0, c[a0.x], v[WGT].y, r0 ; 第2个四元数
mov     a0.x, v[BONE].z
mad     r0, c[a0.x], v[WGT].z, r0 ; 第3个四元数
mov     a0.x, v[BONE].w
mad     r0, c[a0.x], v[WGT].w, r0 ; 第4个四元数

```

正如所讨论的, 规一化混合的四元数后推它为 4D 球体是必需的。

; 规一化四元数

```

dp4     r1.w, r0, r0
rsq     r1.w, r1.w
mul     r0, r0, r1.w

```

这生成了混合的四元数到 r0, 近似 4 个关节上的一个球形插值的加权。

2. 四元数矩阵

为了变换向量, 我们需要转换我们的以内插值替换的四元数为矩阵形式。这种旋转表示为

$$R = \begin{bmatrix} 1-2y^2-2z^2 & 2xy+2wz & 2xz-2wy \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz+2wx \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 \end{bmatrix}, \text{ 其中 } \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} \quad (5.12.3)$$

原生的 C 代码, 看起来如下:

```

Mat[A] = 1.0 - (2.0 * Quat.y * Quat.y);
Mat[A] -= 2.0 * Quat.z * Quat.z;
Mat[B] = 2.0 * Quat.x * Quat.y;
Mat[B] += 2.0 * Quat.z * Quat.w;
Mat[C] = 2.0 * Quat.x * Quat.z;
Mat[C] -= 2.0 * Quat.y * Quat.w;
Mat[D] = 2.0 * Quat.x * Quat.y;
Mat[D] -= 2.0 * Quat.z * Quat.w;
Mat[E] = 1.0 - (2.0 * Quat.x * Quat.x);
Mat[E] -= 2.0 * Quat.z * Quat.z;
Mat[F] = 2.0 * Quat.y * Quat.z;
Mat[F] + 2.0 * Quat.x * Quat.w;
Mat[G] = 2.0 * Quat.x * Quat.z;
Mat[G] += 2.0 * Quat.y * Quat.w;
Mat[H] = 2.0 * Quat.y * Quat.z;
Mat[H] -= 2.0 * Quat.x * Quat.w;
Mat[I] = 1.0f - (2.0 * Quat.x * Quat.x);
Mat[I] -= 2.0 * Quat.y * Quat.y;

```

通过一个基本的编译器运行这段代码会生成一个 25 指令的转换(生成的代码忽略空间)。在这种情况下, 编译器就有可能利用共享积失败, 或者向量化一个单独的运算失败。这就为我们提供了优化的一般起点。

3. 四元数矩阵的优化

我们知道通过四元数转换产生的矩阵是正交的，这样就有一个规范正交基（向量是单位长度并且相互垂直的）。因此，任何单独的行（或者列）能够从其他两个的向量积导出来。根据交叉的顺序，叉积有两个解决方案。坐标系的偏手性决定交叉的正确顺序；其中一个方案对右手坐标系总是正确的，而另一个则对左手坐标系总是正确的。

我们通过交叉基向量 ABC 和 DEF 导出 GHI。或者，

```
; r8 中的 ABC, r9 中的 DEF
mul    r10, r8.yzxw, r9.zxyw ;用交叉得到 GHI
mad    r10, -r9.yzxw, r8.zxyw, r10
```

下一步是一个积极的共享积与共享和的组合。

```
; 四元数 (r5) 到旋转矩阵 (r8,r9,r10) 的转换
def    c[CONST],0.0,1.0,2.0,0.5
add    r6, r5, r5;          2x, 2y, 2z, 2w
mul    r1, r6.xyyy, r5.xyzw ; 2xx, 2yy, 2yz, 2yw
mul    r2, r6.xxzz, r5.ywzw ; 2xy, 2xw, 2zz, 2zw
add    r3, r1.xxyy, r2.zzzz ; (2xx+2zz), (2xx+2zz)
add    r8.x, c[CONST].y, -r3.z; A = 1 - (2yy + 2zz)
add    r8.y, r2.x, r2.w      ; B = 2xy + 2zw
mad    r8.z, r6.x, r5.z, -r1.w; C = 2xz - 2yw
add    r9.y, r2.x, -r2.w     ; D = 2xy - 2zw
add    r9.y, c[CONST].y, -r3.x; E = 1 - (2xx + 2zz)
add    r9.z, r1.z, r2.y      ; F = 2yz + 2xw
mul    r10, r8.yzxw, r9.zxyw ; GHI with a cross
mad    r10, -r9.yzxw, r8.zxyw, r10
```

这种四元数到矩阵转换在 vertex shader 里是 12 条指令。

最终的 shader 混合了 4 个任意关节，将四元数转换到一个矩阵，蒙皮一个顶点，并旋转法线。这个 shader 有 31 个指令长，为一个有 28 块骨骼的角色使用了 GPU 存储器的 56 个四变形寄存器（quad-register）。这可以同顶点混合的实现比较，顶点混合为同样的角色其实现用了 40 条指令和 84 个寄存器[Domine03]。

5.12.4 结论

在这篇文章中，我们用四元数论证了球形关节混合是快速、准确、紧凑的蒙皮解决方案。

5.12.5 参考文献

- [Coxeter73] Coxeter, H. S. M., *Regular Polytopes, Third Edition*, New York: Dover, 1973.
- [Domine03] Domine, Sebastien, "Mesh Skinning," available online at <http://developer.nvidia.com/object/skinning.html>, July 1, 2003.
- [Mohr03] Mohr, Alex, and Michael Gleicher, "Building Efficient, Accurate Character Skins

from Examples,” ACM SIGGRAPH 2003.

[Shoemake85] Shoemake, Ken, “Animating Rotations with Quaternion Curves,” ACM SIGGRAPH 1985.

[Shoemake92] Shoemake, Ken, and Tom Duff, “Matrix Animation and Polar Decomposition,” *Proceedings of the 1992 Graphics Interface Conference*, pp. 245–254, 1992.

[Shoemake94] Shoemake, Ken, *Quaternions*, May 1994, available online at <ftp://ftp.cis.upenn.edu/pub/graphics/shoemake/quatut.ps.Z>, July 1, 2003.

[Weber02] Weber, Jason, “Improved Deformation of Bones,” *Game Programming Gems 3*, Charles River Media, 2002.



5.13 动作捕捉数据的压缩

作者: Søren Hannibal, Shiny Entertainment

E-mail: sorenhan@yahoo.com

译者: 刘永静

审校: 谷超

在电视游戏领域, 用于游戏研发的资金每一年都在增长, 预算已经超过了上千万美元。通常, 预算的大部分是为动作捕捉阶段保留的, 它会产生几十亿字节的原始关键帧数据。然而, 由于在游戏开发中游戏机的内存仍然是一种稀缺资源, 因此折衷的结果往往导致大量有趣的动画被删掉, 仅仅保留那些占内存最少的部分。

这篇文章示范了一个有损耗的压缩系统, 它被开发出来用于提高动作捕捉数据的内存利用率。虽然系统有一些特性利用了骨骼层级, 但是可以在任何一个关键帧动画上使用更普通的解决方案, 例如事先录制的摄像机移动和手工活动物理对象。

本文集中在动画数据的压缩和解压缩上, 它跟模型如何被渲染、动画如何被混合在一起、曲线如何被内插进去无关。这里介绍的技术可以很容易地在现有的动画系统中实现。

内存的节省取决于应用压缩的积极程度。一个具有 16 块骨头的分层级模型的动画, 其大小可以被压缩到低于 500Kbytes, 而未压缩的数据则是每秒超过 15 000Kbytes。解压缩的性能花费可以忽略不计。压缩是处理器加强器 (processor-intensive), 为了得到一个最佳的结果, 它需要少量的用户交互。

5.13.1 处理的计划

压缩计划有 3 个步骤:

- (1) 组织数据以至于存储最少量的数据通道;
- (2) 通过移除那些对动画影响很少或者没有影响的键来减少键的数量;
- (3) 通过降低精度来包装剩余的键。

以上每一步都有一些对动画有用的设置。有一些经验的动画师会选择一些设置, 以实现通过最少的调整来产生最好的结果。因此, 保持设置的数量低一点很重要, 免得吓退动画师。

5.13.2 组织数据通道

第一步是计算出哪个数据通道是必需的。鉴于这个系统的目的，数据通道被定义为一系列可旋转的、可移动的，或者可放缩单个骨骼的键。对一个标准的人体，你需要储存根部的位置和每个骨骼的方向。然而，当系统必须支持面部动画、手部动画、挤压和伸展动画，或者必须渲染其他类型的有关节模型如汽车或者武器时，在所使用通道的选择中，对其弹性的考虑变得非常重要。

同样地，如果系统必须在一个角色上同一时间播放多种动画，那么你或许不想为所有的动画计算所有的骨骼。一个角色在奔跑的同时瞄准武器就是这样的一个例子。角色将使用标准的大腿和躯体奔跑动画，同时使用手臂和肩膀紧握武器的动画。

如果不需要绕所有三个轴全部都自由旋转，那么一些通道可以被进一步的优化。例如，下颌骨、眼睑和一些手指骨仅需绕单个轴旋转。通过只存储单个轴，你可以节省大量的空间，这样一来你只需要存储 1/3 的数据。由于从单个轴创建局部矩阵比从三个轴重建它或者从一个四元数中转换它更快，因此速度也提高了。在进行键间内插入或者动画间混合时，一个单轴内插值也比一个四元 `slerp` (spherical linear interpolation, 球面线的内插值) 更简单、更快。

注意，对一些动画来说，一些关节可能从单轴变为多轴。例如，如果下巴被限定在仅能绕单轴旋转，那么一个显示角色将她的下巴从一端移动到另一端的动画将不可能实现。

5.13.3 减少已储存的键的数量

一旦你组织完所有的数据，那么它就必须被优化，以便可以移除那些只能看到很少或者根本就看不到的键。

此算法本身很简单：

- (1) 测度每一个键的重要性；
- (2) 根据一些标准移除一些键；
- (3) 移除最不重要的键；
- (4) 重新计算与被移除的键相邻的两个键的重要性。

虽然算法非常简单，但是如果没有按照下面两个问题组织数据，那么它运行起来会非常慢。这两个问题就是：1) 哪个键是最不重要的？2) 对每个剩余的键来说，哪两个键同它相邻？

对每个通道来说，一组键按照帧编号的分类被储存。这些键也应该在一个连接列表中被连接在一起。一个键无论何时被移除，它都将从连接列表中被取出；因此，我们不仅可以知道哪些键是可用的，哪些键已经被丢弃，还可以很快地找到每个被移除键的相邻键，这在查找需要被重新计算的键的时候是很有用的。

它也对具有一个平衡的，按照在所有通道中键的重要性分类的二叉树有帮助。这将充分减少它花费在查找最不important键和在键的重要性被重新计算后在二叉树中重新配置键上的时间。

1. 测度每个键的重要性

可以导致压缩系统成功或失败的一件事情是对测度每个键的重要性的探索。这种探索跟压缩的水平下降没有太多关系，但是当使用更高的压缩率的时候，它变得越来越重要。这可以解释，在一个不自然地挥动着双翼的3D模型，和一个活的、有呼吸的人之间的差异。

动画师应该可以单独地调整为每个动画移除的键的数量。这样就可以输入在动画的每一秒所需移除键的目标数值。在每个动画上使用的最佳压缩数量是不一样的，但是经过一番很好的探索，每秒钟5~10个键可以制作出正常的动画。

最简单的探索是模拟从最优化的数据集中对每一个键的移除并比较移除前后的数据集。通过解压缩两个数据集中的帧并对它们进行比较，可以完成这个模拟。

由于很多键处于同一直线，因此根据原始数据集来查找键的重要性是不可接受的。共线性导致的问题是，如果处于同一直线上的任何键被移除，那么其他的键将通过和以前一样的路径调节曲线。可是，如果所有共线的键都被移除，那么曲线将改变它的路径。

对在游戏中计算键的重要性和解压缩帧来说，使用相同的内插值计划是很重要的。线性内插法会带来最差的结果，因而不推荐使用。使用线性内插法的动画看起来常常像机械般很不自然的。样条内插法（Spline interpolation）通过平滑动画可以产生更好的结果。

2. 为键的移除找到正确的标准

可能存在很多不同的移除标准。最明显的标准是测度两个数据集中的键离的有多远。例如，通过使用毕达格拉斯（Pythagoras）的方法可以实现这样的标准。使用标准的时候，所有被解压缩的帧将越来越像原始动画的帧。

然而，为了让动画更像现实，个别帧跟最终的动画的总体感觉毫无关系。如果探索只考虑绝对值，那么每一帧看起来都是正确的，但是当在动作中被观看时，这些帧可能远远偏离了原始动画。速度和加速度是两个同样重要的标准，它们是曲线的第一派生事物和第二派生事物。例如，冲压机动画需要有正确的加速和减速以体现快速的，受控制的感觉。虽然电视游戏玩家不知道在原始的动作捕捉中冲压机下落的地点，但是当他们玩游戏的时候，如果感觉动画不对，他们就会注意到。

当处理分层级动画的时候，每个骨骼的位置关系非常重要。转体动作影响物体的外延要比物体的中央更多。手关节中的1度差异比肩关节上的1度差异更不被人注意，因为在肩膀和指尖之间的骨骼数量要比手和指尖之间的骨骼数量更多。这是一个不应该被自动决定的地方，因为在某些情形中，动画师想把焦点放在身体的某些部位。例如，如果有一个手的特写镜头，那么在手骨骼和根骨骼之间的所有骨骼都使用高质量存储是很重要的，相反手臂和腿则无关紧要。因此，一个能达到较好结果且容易的方法就是让动画师来控制身体的哪组需要更多的细节。

最后，如果骨头在某个帧上存储了一个键，那么子骨骼和父骨骼应该在同一个帧上存储一个键。这倾向于创建一个平静的且更高精度的动画，特别是在使用现行内插值的时候。

3. 重新计算相邻键的重要性

键的移除对它的相邻键有影响，它们的重要性需要被重新计算以便去除共线性的问题。

如果你使用样条内插值，那么在被移除的键的每一边的两个剩下的相邻键都会受到影响。如果使用线性内插值，那么两个剩下的相邻键中则只有一个相邻键会受到影响。

如果探索将父骨骼或子骨骼考虑在内，那么那些可以影响骨骼的键也需要重新计算它们的重要性。

5.13.4 包装剩余的键

困难的部分已经解决了。现在你已经找到了需要存储的键，是考虑如何包装数据的时候了。不是用 32 位浮点格式存储每件事物，而是用更小精度存储最终值来完成重要的内存节省的工作。

把四元数打包成 4 个字节[Zarb-Adami02]的结果相当好。可以为位置动画开发相似的方法。这种类型的技术在快速移动的动画上工作特别好，但是在玩家可以很容易注意到角色的细节的缓慢动画上，它将造成动画抖动。

在分层级的系统里，大部分的抖动是由靠近角色根部的骨骼造成的。因此，如果你用不同的精度级别包装不同的通道，那么动画将可以受益。表示靠近根部的骨骼的键应该使用 8Byte 或者 12Byte 的精度被储存，而表示手臂和腿的键则使用 4Byte 的精度被储存。

很明显，每个动画可以有不同的包装级别。如果你知道某个动画是在特写镜头中被使用，或者一个看起来不是很好的动画，那么你就可以使用最高质量的包装。此外，这应该是动画师能够调节的东西。允许动画师调节这个值的一个容易的方法是有大量的预定义设置。

键数据被包装后，在压缩过程中只剩下一件事情，那就是使用一个合理的格式来存储数据。对每个通道来说，你需要存储剩下的键和它们的帧编号。为了实现内存使用最小化，你可以为每个键使用多字节格式存储作为先前剩余键的偏移量的帧编号。这意味着使用了 1Byte 或者 2Byte，第 1 个字节的高位将指示第 2 个字节是否被使用。使用这个方法的动画不能多于 $32\,768\ (2^{15})$ 帧。

5.13.5 运行时解压缩

渲染每一帧的解压缩需要两步：

- (1) 更新解压缩缓冲，以便键在正确的帧范围；
- (2) 在解压缩缓冲中，为了得到帧的动画，在键之间内插值。

在每一个动画的开始，动画播放器必须为每个数据通道分配一个解压缩缓冲。这个缓冲用来存储那些可以让内插值程序得到当前帧的未包装键。每一个解压缩缓冲应该足够大，可以容纳每个数据通道的 2 个或者 4 个未压缩键，它取决于在压缩期间使用的内插值的类型。在整个动画期间，这些缓冲保持在内存中，当它对当前的动画帧无效的时候，每一个缓冲将被更新。

最后一步是在键之间内插值。这可以用和在压缩期间使用的相同的内插程序来完成。这一步完成之后，你就全部做完了。你可以成功地解压缩动画的每一帧。到那时候，剩下的事情就只有渲染模型了。

5.13.6 未来的改进

如果你理解了前一部分中的步骤，那么你就拥有了一个完整的压缩系统，具有极佳地递送良好的压缩过的动画的能力。然而，就像其他每件事情一样，有很多的方法可以改进这个系统。虽然你自己或许也可以提出一些好主意，但是这里有一些建议和技巧。

- **曲线拟合。**很多键被移除后，曲线或许不再是最佳了。曲线通过了存储键的帧的值，但是在键之间，曲线将消失。通过试着使曲线成为对所有组合的键来说都是最佳结果，你将能够让全部的动画接近于原始动画，同时仍然保持时间未被改变。

- **存储交叉存取的通道。**如果动画通道被交叉存取的储存，它可能在动画中在某个时间涌出一小片。这对削减场景来说是有用的，因为在内存中不需要立刻有一个完整的动画。规则的动画也可以从中受益，因为它将保持内存的随机存储，这是由于内存中只有一个位置可被读取，而不是每个通道一个位置。

- **从键包装中分离键移除。**键移除是系统最慢的部分，但是它只是动画需要调整的部分的其中之一。通过存储键移除和键包装之间的中间结果，动画或许可以调整包装系数并且可以比必须从开始重新压缩动画的系统更快地看到结果。

5.13.7 结论

每个游戏都有内存约束。压缩动画数据是你收回一些内存的一种途径。压缩是一个简单的过程，它简单地移除那些对动画的外观和感觉不重要的键，并且尽可能的包装剩余键。

测度键的重要性的探索是最重要的，它是得到正确结果的最困难部分。然而，即使是精确度较低的探索都可以节省很多内存。打包剩余键也是很重要的，在高压缩级别下，由包装过的键引起的抖动消失了，使结果看起来比未压缩的使用包装过的键的动画更好。

从一个动画到另一个动画，结果都不一样。没有一个好的选项集可以被普遍的应用。因此，让动画师可以自由地为每个动画调节压缩级别是很重要的。然而，为了使动画师能够快速处理大量的动画，将选项的数量削减到最小是很重要的。

如果不动手，它看起来或许很复杂，但是你只要编写一个压缩程序就全懂了。记住，规则不是刻在石头上的，而是拿来用作方针的，有空就去做实验吧。

5.13.8 参考文献

[Zarb-Adami02] Zarb-Adami, Mark, "Quaternion Compression," *Game Programming Gems 3*, Charles River Media, 2002.

5.14 基于骨骼的有关节的 3D 角色的快速碰撞检测

作者: Oliver Heim、Carl S. Marshall、Adam Lake, Intel Corporation

E-mail: oliver.heim@intel.com,
carl.s.marshall@intel.com,
adam.t.lake@intel.com

译者: 刘永静

审校: 谷超

模拟是任何一个以实现让用户沉浸在虚拟世界中所必需的现实性为目的的游戏引擎的重要组成部分。对现实的虚拟体验的追求需要高性能的 CPU、图形卡、带宽和内存。可推动性能提高的最可计算的加强器领域之一是正确而高效的碰撞检测的应用。使用围绕一个精灵的矩形作为一个模型的近似值的日子一去不复返了。我们需要看起来和感觉上都正确的, 发生在 3D 空间中的碰撞。本文的目的是给出一个实用高效的 3D 游戏引擎的实现, 它具有有一些比在当今引擎中使用的普通技术更好的优点。

5.14.1 碰撞检测与碰撞分解

我们将碰撞检测定义为查找场景中的两个物体相互碰撞的瞬间, 将碰撞分解定义为通过运用一些跟碰撞相关的对象的作用力和反作用力来解析碰撞的作业过程。移动物体使它们的三角形不相交, 在对象上应用基于物理的影响力或者选择两个物体中的一个来应用反作用力, 这些都仅仅是分解碰撞的一些方法。碰撞分解本身是一个很大的主题, 本文主要集中在碰撞检测。

5.14.2 术语

骨骼: 一个有骨架角色的每一块骨骼跟一系列的顶点相关。当骨骼变形的时候, 顶点连同骨架一起移动。在碰撞检测中, 我们使用相同的骨骼作为输入值提供给碰撞检测子系统。一个顶点可能跟几个骨骼相关, 并且每个骨架有一个权重因数, 这是它的一个特色。此外, 骨骼通过一种父子关系被连接起来, 组成了模型的骨骼层次或骨架。

AABB: 轴向排列边界盒 (Axis-Aligned Bounding Box)。盒子的每个面都按照某个主轴排列。

OBB: 面向边界盒 (Oriented Bounding Box) [Gottschalk96]。OBB 是一种广义性的 AABB，它可以在 3D 空间中被任意旋转。因为盒子的旋转是仿射的，所以在旋转的时候，这些盒子不能伸长或者缩短。由于每个 OBB 表现为跟骨骼相关，因此当骨骼变形的时候，我们不需要重新计算 OBB 的位置和方向（见图 5.14.1）。

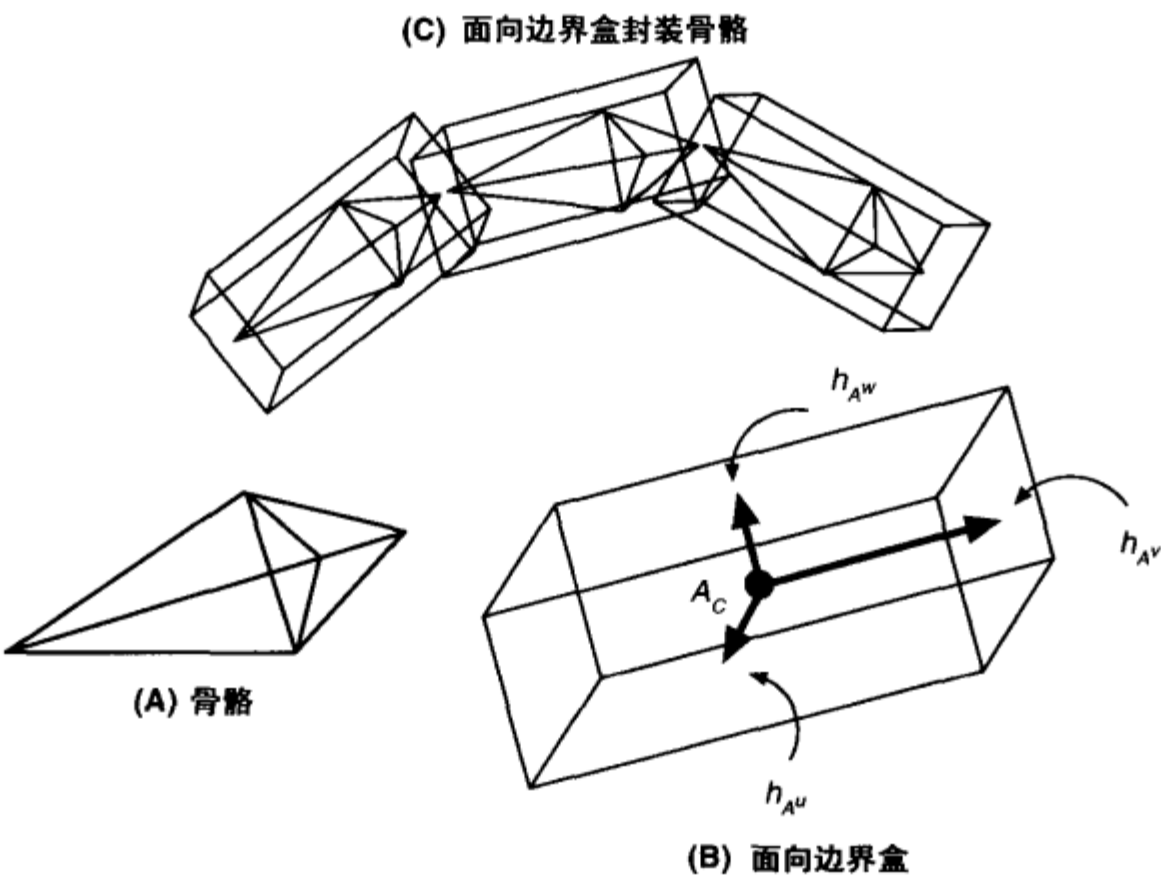


图 5.14.1 一块骨骼，一个面向边界盒和围绕三块相互连接的骨骼的面向边界盒。 A_c ，表示盒子的质心（近似中心）， $\{h_{Aw}^A, h_{Av}^A, h_{Au}^A\}$ 表示正的盒子宽度的一半

蒙皮: 对某个网格模型信息的处理，例如顶点，三角形和顶点权重，都被映射到基于骨骼模型的根本的骨架。每个顶点可以被映射到骨架中的 1 个或者多个骨骼，相应的重量决定了当一个顶点的根骨骼被移动的时候，它要被转换多少。因此，一旦骨架被蒙皮，当骨架的方向发生变化的时候，网格模型可以自然地发生变形。由于蒙皮跟快速碰撞检测算法的运行时不直接相关，因此我们在决定哪些顶点/三角形跟每个骨头相关的预处理阶段应用蒙皮运算。

5.14.3 将碰撞检测集成到 3D 游戏引擎中

为基于骨骼的角色进行碰撞检测是一个计算的精深的问题，因为它涉及到两个不同的转换阶段。在模拟阶段，我们将所有的顶点转换到碰撞检测引擎的坐标系统中。在渲染阶段，为了进行蒙皮运算，所有的顶点再一次被转换。在碰撞检测期间一次性完全转换是行不通的，但是我们需要尽量减少这种计算。

在我们开始钻研被提议的解决方案的细节之前，先让我们看看在一个游戏每一帧的要素上可能会发生什么。此外，我们假设运行每一帧的游戏引擎是一个循环。

For 每一帧:

```
While 剩余的模拟时间:
    将系统提前到当前时间
    For 每个模型:
        将模型转换到世界空间
        在基于骨骼模型中转换骨骼
    DetectCollisions();
    ResolveCollisions();
    SkinBonesBasedModels();
    RenderScene();
```

从前面的伪代码可以看出,在产生最终显示的图像的过程中包括几个脱节的步骤。例如,在碰撞检测之前渲染帧是不可取的,正如在下一系列的世界空间转换被应用到每个模型之前进行碰撞检测是不可取的一样。

传统的碰撞检测解决方案

直接了当的解决方案是使用一些互相分离的网格拓扑,一个被碰撞检测使用,另一个被渲染使用。在这种情形下,碰撞网格模型的分辨率一般比渲染的网格模型更低。这种解决方案可以工作,但是隶属于一个给出了不同拓扑的不正确的碰撞结果。更重要的是,虽然整个网格模型仍然被转换,但是转换后的网格模型可能更小了。然而,碰撞领域或许只是网格模型中多边形总量的一个小子集。

另一个解决方案是使用 proxy 或者 impostor 场景,就像一个边界体积层级 (bounding volume hierarchy),在对碰撞的检测上,它比较接近模型的场景。一个边界体积层级组织了一个类似树结构的场景,在这个场景中每一个更低的级别把父级别分成大量预先确定的子体积 (subvolume)。一旦完全地完成组装,具有 N 个多边形的模型树就包括一个包装了完整模型的单一根体积和若干个内部的级别,在最低级的级别处有 N 个叶子节点。在每个叶子节点,有一个单一三角形附在边界体积中。通过使用树结构,边界体积层级可以产生更快的碰撞检测,因为在任何一个给定的碰撞检测测试中,树中的大多数边界体积都不需要交叉测试。由于层级的计算十分昂贵,因此它仅适合静态模型,在静态模型的生命周期内,它被创建一次就可以重复使用。对于动态模型例如基于骨骼的角色来说,它是非常不理想的,因为它必须每隔一定时间就要重新计算模型的方向或者位置的变化。

5.14.4 基于骨骼的快速碰撞检测算法

我们的算法使用模型的骨骼来实现物体的快速碰撞检测。我们通过计算有关节模型的每一根骨骼的简单的边界体积来实现这个算法,而不是为整个模型提供一个完整的边界体积层级。在骨头列表中的每一根骨骼都有一组与之相关的顶点;因此我们只需要转换碰撞中跟边界体积相关的顶点,就可以快速地确定哪里发生了碰撞。在接下来的几个部分中,我们要描述一个数据结构、预处理步骤和算法的运行时。

上述的碰撞检测算法所包括的优点:

- 只在有需要的顶点上应用转换;
- 除非骨骼尺寸改变,否则 OBB 不需要重新计算;

- 不需要在每帧的元素上重新计算边界体积层级，因为每根骨骼的体积是静态的；
- 不需要有不同的模拟与渲染拓扑。

1. 使用面向边界盒实现碰撞检测

我们的算法的核心是基于 Stephan Gottschalk 的面向边界盒 (oriented bounding box) 交叉测试[Gottschalk96]。这种算法使用分离轴定理 (separating axis theorem)，该定理表述的是通过充分的查找一个在 OBB_A 和 OBB_B 之间的单轴来证明它们是脱节的。最多需要测试 15 个轴：来自 OBB_A 面的 3 个轴，来自 OBB_B 面的 3 个轴，来自 OBB_A 和 OBB_B 边的组合的 9 个轴。所有的测试包括投射每个盒子半径的总数到被测试的轴上。如果所有 15 个测试轴的投影都交叠，那么就会产生一个交叉点。图 5.14.2 呈现了一个单一投影的 2D 版本，它显示这些 OBB 都是脱节的。要得到这方面运算的更多的信息，请参见[Gottschalk96]。

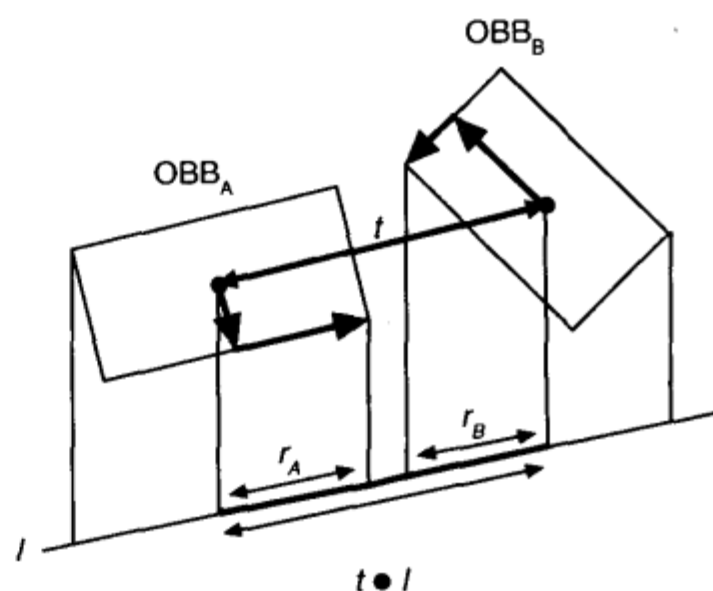


图 5.14.2 分离轴图解——图形显示， OBB_A 和 OBB_B 是脱节的，因为轴 l 上的 radii, $r_A r_B$ 的投影是不交叠的

2. 数据结构

在开始钻研核心算法之前，了解我们的数据结构如何定义是很重要的。因为我们选择面向边界盒作为代理几何体，所以数据结构将反映这一点。下面是我们的算法所使用的最基本数据结构的部分清单。

```
class OrientedBone
{
    // 骨骼信息
    Matrix4x4    m_mReferenceTransform;    // 骨骼 xform
    Int          m_iID;                    // 骨骼 id
    TriList*     m_pTriangleList;          // 每一骨骼的三角形
    Vector3*     m_pVertexList;            // 顶点索引

    // 面向边界盒信息
    Vector3      m_vCentroid;
    Vector3      m_vAxisU, m_vAxisV, m_vAxisW;

    // OBB 的正半宽
    float        m_fHalfWidthU, m_fHalfwidthV,
                m_fHalfWidthW;
};

class BoneModel
{
    // 指向骨骼层级的指针
```

```
OrientedBone**    m_pBoneList;

// 包含球形的位置和{h}半径的向量
Vector4           m_vBoundingSphere;

// 整个模型的世界空间位置/方向
Matrix4x4         m_mPreviousOrientation,
Matrix4x4         m_mCurrentOrientation;
Matrix4x4         m_mNextOrientation;
};
```

3. 预处理

预处理阶段由两个步骤组成：从 3D 认证包中读取输出场景信息和为快速的基于骨骼碰撞检测算法设置数据结构。当遇到一个包含骨架的模型时，我们读取模型的原始位置和方向，还有组成骨架的骨骼数量。然后创建一批等于模型骨架中骨骼数量的 `OrientedBone`。接着，骨架中的每块骨骼被处理；换句话说，我们从场景文件中读取特定骨骼的信息，包括一个惟一的骨骼标识符，它的父骨骼的标识符和提供它相对于根骨骼的位置和方向的变换式，填充到与骨骼相关的数据结构中。每根不同于根骨骼的骨骼只有一个父亲节点，可以具有 0 个或者多个孩子节点。使用一个预定顺序的往返移动来处理骨骼，用一个递归的方式，从根骨骼开始向外朝叶子骨骼移动，并将惟一的骨骼标识符插入到骨骼列表中。一旦到达了一个叶子骨骼，我们就再从根骨骼开始，直到所有的骨骼用相同的方式被处理完毕。

当所有的骨骼被处理完毕，我们读取跟模型相关的网格模型信息。它包括顶点、三角形和骨骼重量，它们是使骨骼的骨架蒙皮和为数据结构进行边界体积（`bounding volume`）的计算所必需的。我们必须在确定哪个顶点跟骨架中的每块骨骼相关的预处理期间执行一个单一的蒙皮运算。每块骨骼的结果顶点列表随后被用于计算一个完全围绕骨骼和它的顶点的面向边界盒。由于这个计算发生在模型空间中，因此我们可以使用标准的轴向排列边界盒技术来计算面向边界盒——我们只需要沿着每个主轴简单地查找最小和最大值。虽然这种技术可能不能产生一个最小体积的边界盒，但是它可以提供一个极其合适的值。要得到关于为给定的一组顶点计算最小边界体积的更多信息，请参见[Eberly01] 或者 [Gottschalk96]。完成之后，我们有了一列骨骼，它们按照惟一的骨骼标识符排序，并且包含每一块骨骼的面向边界盒，相关的顶点和三角形。

4. 运行时

基于骨骼的模型在运行时有两个不同的阶段：模拟阶段（骨架移动和碰撞检测等等）；渲染阶段，在其中模型被蒙皮。在本部分中，将要讨论我们的算法中属于模拟阶段的方面。

一个游戏引擎的模拟阶段一般由一个调度程序控制[Harvey02]。通过分配时间段到不同的模拟成分，调度程序确保没有一个单独的成分可以消耗所有的系统的资源，因此保持了玩游戏的平稳流畅。为了使碰撞检测更加容易，调度程序提供了一个碰撞引擎，它是带有一个标注有起始时间和结束时间的时段的一系列碰撞模型。起始时间表示每个模型的当前世界空

间位置和方向，而结束时间则表示一旦所有的碰撞被检测到并解决掉时每个模型的未来世界空间位置和方向。

碰撞引擎首先计算每个模型的新世界空间位置和方向。如果模型包含骨骼，每块骨骼的新世界空间位置和方向也必须被计算。其次，在每对模型之间完成成对的交叠测试以确定是否存在一些交叉点。考虑到效率，每个模型要符合边界球体，以便只有潜在的碰撞需要额外的校验（参看图 5.14.3）。

```
DetectCollisions()
{
    while( FrameTimeLeft )
    {
        // 提前计算下个时间世界空间
        // 位置、方向和骨骼（如果有的话）
        AdvanceModels(nextTime);

        for each model pair - modelA, modelB
        {
            collision = false;

            if( SphereIntersection(modelA, modelB) )
            {
                if( modelA->HasBones() ||
                    modelB->HasBones() )
                    collision = OBB_Bone_Traverse();
                else if( modelA->HasBones() &&
                        modelB->HasBones() )
                    collision = Bone_Bone_Traverse();
                else
                    collision = OBB_OBB_Traverse();
            }

            if( collision &&
                collisionTime > firstCollisionTime )
            {
                SaveModelInformation();
                firstCollisionTime = collisionTime;
            }
        }

        if( collisionOccurred )
        {
            ResolveFirstCollision();
            nextTime = collisionTime;
        }
        else
            nextTime += Step;
    }
}
```

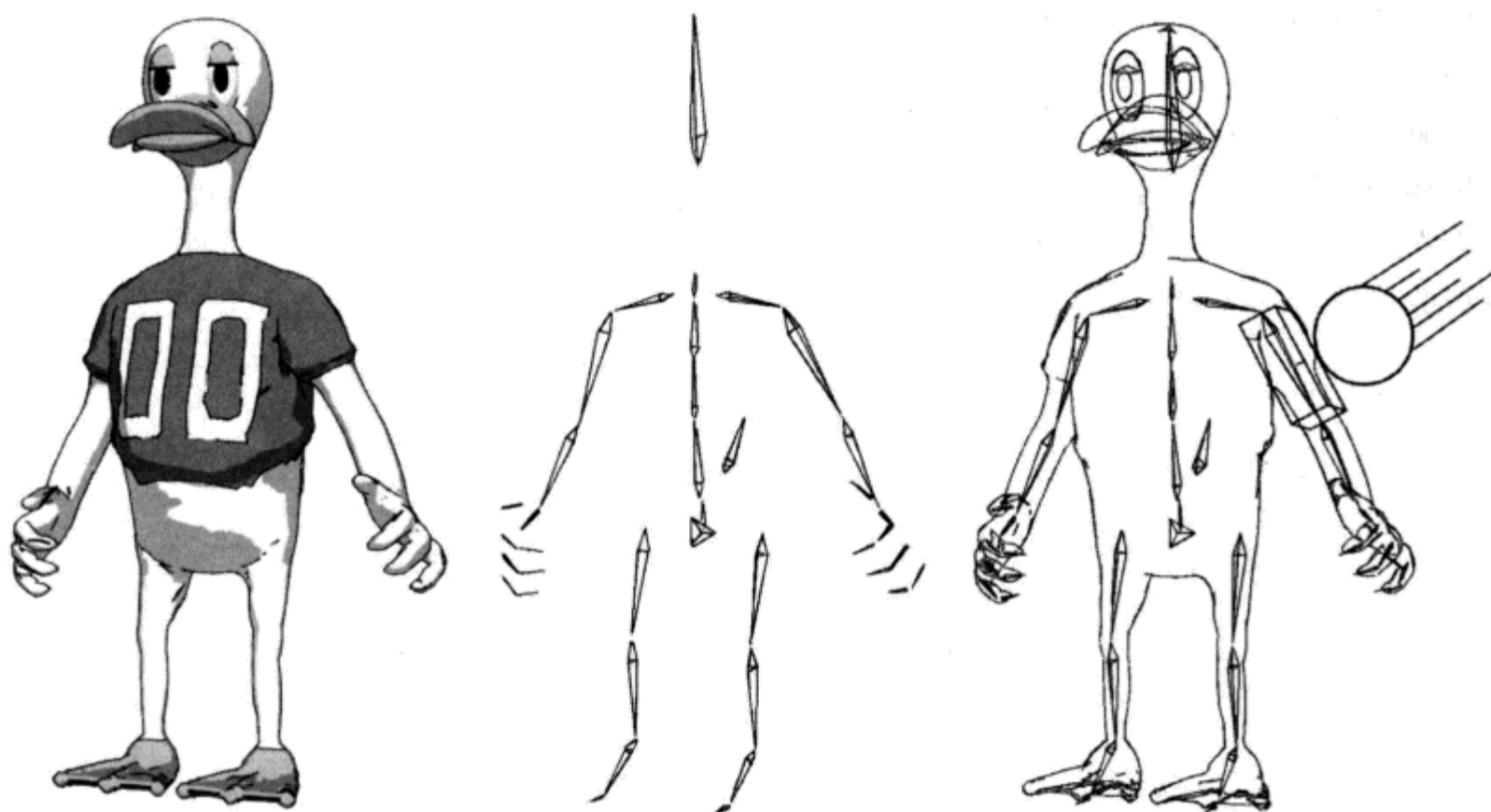



图 5.14.3 (A) 显示的是鸭子的 3D 模型。(B) 显示的是能够让鸭子动起来的基于骨骼的层级。
(C) 显示的是鸭子里面的骨骼层级，并且鸭子的左上臂受到一个小球的撞击

一旦发现了两个交叠的球体，我们就必须校验是两个潜在交叠模型中的其中之一还是两个都包含骨骼列表。如果两个都没有，那么我们就进行标准的 OBB 往返移动/交叠测试。否则，我们就必须测试与无骨骼模型的 OBB 层级相反的骨骼模型，或者测试两个骨骼模型有无交叠。在后面的情形中，我们简单地比较了所有模型 A 和模型 B 的面向骨头。如果有一对交叠的盒子被发现，那么我们就必须转换每个跟骨骼相关的顶点并且在多边形级别对交叉点进行测试。

```
void Bone_Bone_Traverse( BoneModel *pBoneModelA, BoneModel *pBoneModelB )
{
    OrientedBone** ppBoneListA= pBoneModelA->GetBoneList();
    OrientedBone** ppBoneListB= pBoneModelB->GetBoneList();
    int i, j;

    for(i=0; i<pBoneModelA->GetNumBones(); i++)
    {
        for(j=i; j<pBoneModelB->GetNumBones(); j++)
        {
            if( ppBoneListA[i]->OBBIntersection(
                ppBoneListB[j]);
            {
                ppBoneListA[i]->TriangleIntersection(
                    ppBoneListB[j]);
            }
        }
    }
}
```

我们感兴趣的其他交叉点类型是一个骨骼模型和无骨骼模型的交叉点。在这种情形下，我们必须测试模型 A 的骨骼列表中并且与模型 B 的 OBB 层级相反的每块骨骼。如果有一个这样的交叉点被发现，那么我们就继续测试与模型 B 的 OBB 层级更低级别相反的骨骼。请注意只有在到达一个叶子节点后我们才能转换与那块骨骼相关的一些顶点，从而防止大量的无用顶点转换，这一点很重要。这个时候，我们必须测试转换后的与骨骼相关的多边形是否与同模型的叶子节点相关的多边形相反。这个测试在世界空间中执行，因此可以获得并返回给用户正确的碰撞信息。

```
void OBB_Bone_Traverse(OBB *pBox, OrientedBone *pBoneBox)
{
    if( pBox && pBoneBox )
    {
        // 测试 pBoundA - pBoundB 的交叉
        bool bOverlap = pBox->OBBIntersection(pBoneBox);

        if( bOverlap )
        {
            OBB* pRightChild = pBox->GetRightChild();
            OBB* pLeftChild = pBox->GetLeftChild();

            // 检查是否已经到达了叶子节点
            if( !pRightChild && !pLeftChild )
            {
                // 我们已经到达了叶子节点
                // 1. 转换 pBox 的三角形
                // 2. 转换 pBoneBox 的三角形

                pBox->TriangleIntersection(pBoneBox);
            }
            else
            {
                // 向下递归 pBox 的左边分支
                if( pLeftChild )
                    OBB_Bone_Traverse(pLeftChild, pBoneBox);

                // 向下递归 pBox 的右边分支
                if( pRightChild )
                    OBB_Bone_Traverse(pRightChild, pBoneBox);
            }
        }
    }
}
```

关于每个碰撞的信息，包括接触的近似点、每个模型的位置/方向、在接触点的法线向量和碰撞的时间，都按照时间顺序存储，直到在这个时间段的所有的碰撞被发现。这时，一个回调程序通知用户发生了一次碰撞，需要进行碰撞分解。请注意在每个时间段只有第一次碰撞（如果同时发生两个或者更多，则是一些碰撞）需要被分解。一旦碰撞被分解，系统就被更新以反映当前时间的状况，如果还剩下一些帧时间，那么我们继续寻找更多的碰撞。当帧

时间结束的时候，我们就把控制权交回给调度程序。

5. 分析

既然我们已经介绍了快速碰撞检测算法是如何工作的，那么让我们来看看这种算法能比传统的方法（没有边界体积）和完全的边界体积层级方法节省多少实际消耗。我们可以使用一个消耗函数[Moeller02]来检查每种方法的性能，它可以量化每种方法所需的计算来完成交叠测试和边界体积（BV）更新。

$$t + n_v c_v + n_p c_p + n_u c_u$$

- n_v : BV/BV 交叠测试的数量。
- c_v : BV/BV 交叠测试的花费。
- n_p : 最初成对的交叠测试的数量。
- c_p : 最初成对的交叠测试的花费。
- n_u : 在模型运动期间已经更新的 BV 数量。
- c_u : 更新 BV 的花费。

让我们假定，对蒙皮而言，每一个有两只脚的鸭子模型包含 30 块骨骼和 1000 个三角形，而球状物则包含 100 个三角形。对于传统的情形，我们假定每个模型的低分辨率网格包含 500 个三角形，并且其中的 50 个三角形，每个三角形包含一个围绕整个模型的单一 OBB。此外，对于全部边界体积层级的情形，鸭子的层级是平衡的，为 8 级深，生产 128 个 OBB；而球状物的层级为 5 级深，生产 16 个 OBB。最后，我们假设给出一个平衡的树，每个叶子节点包含 8 个三角形。为了得到一个定量的比较，我们必须考虑精确的近似值和为边界体积计算和交叠测试而执行的比较操作。

- OBB 创建 – 486 ops
- OBB 交叠 – 210 ops
- 三角形交叠 – 80 ops

结果如表 5.14.1 中所提供的那样。从结果来看，我们所描述的快速碰撞方法比其他两种方法可提供大量更高的性能。

表 5.14.1 算法分析

算法	nv	cv	np	cp	nu	cu	总操作
传统的	1	210	12 500	80	0	486	1 000 210
完全的边界体积层级	25	210	32	80	128	486	70 018
快速碰撞	200	210	120	80	0	486	51 600

对那些不包括骨架的对象来说，全部的 OBB 层级将提供最好的性能，这是显而易见的，因为层级在预处理期间只要创建一次就可以在每一帧中重复使用。然而，以相同的方式，OBB 的创建是一个昂贵的运算，并且当它为了基于骨骼模型必须被重新创建在每一帧（或者更不好是，在子帧上）的要素上时，就不能够得到最佳的性能。最后，虽然为潜在的碰撞测试每一块骨骼是昂贵的，但是我们也节省了每一帧更新边界体积的花费。从整体上说，这种方法可以节省大约为 26%的花费。

6. 优化和未来的工作

虽然我们还没有实现或者测试其他的边界体积类型，但是为每一块骨骼使用不同的边界体积会有一些优点。例如，对每块骨骼来说，边界体积圆柱体可能跟 OBB 一样好。此外，在测试每一块骨骼之前平衡在角色中的边界体积层级的一些其他类型也能得到一些好处。最后，更高级选择装置的使用也可以改善整体的性能，此选择装置用于减少必须在每一个时间段被校验的成对碰撞的数量。

5.14.5 结论

这篇文章描述了如何在即时游戏引擎中进行现实的且优化的碰撞检测。下一次你在创建现实的基于骨骼角色动画时，就可以进一步通过基于骨骼的碰撞检测来提高你在场景中的角色的运动。这种技术最有价值的一方面是其平衡角色动画必需的当前基础构造的能力。

5.14.6 感谢

我们要感谢 Intel/G3D 小组以及他们在 Shockwave3D 引擎的创建上所做的工作。此外，我们还应该感谢 UNC-Chapel Hill 的 Dinesh Manocha 和 Ming Lin 在碰撞检测领域的协作和所做的额外工作。

5.14.7 参考文献

[Eberly01] Eberly, D., *3D Game Engine Design*, Morgan Kaufmann, San Francisco, CA, 2001.

[Gottschalk96] Gottschalk, S., M. Lin, and D. Manocha. "OBBTree: A Hierarchical Structure for Rapid Interference Detection," *Proceedings of SIGGRAPH 1996*, pp.171-180.

[Harvey02] Harvey, Michael, and Carl S. Marshall. "Scheduling Game Events," *Game Programming Gems 3*, Charles River Media, 2002.

[Moeller02] Moeller, T., and E. Haines. *Real-Time Rendering, Second Edition*, A.K. Peters Ltd., Natick, MA, 2002.



5.15 使用地平线进行地形的遮挡剔除

作者: Glenn Fiedler, Irrational Games

E-mail: gaffer@gaffer.org

译者: 刘永静

审校: 谷超

本文描述了一种以高度场地形几何学 (heightfield terrain geometry) 为基础的对于室外场景的遮挡剔除 (occlusion culling) 技术。不像其他的地形遮挡剔除 (terrain occlusion culling) 技术, 它不需要昂贵的离线处理, 因而它能够适合动态改变的地形。

5.15.1 引言

如果游戏里的一个玩家站在山脚下, 那么从视线上来看, 山后的很多对象就被遮挡住了。既然这些对象不能被看到, 那么就绝对没有必要渲染它们。如果我们能够很快地检测并剔除任何被山所遮挡的对象, 那么被渲染对象的数量就大大地减少了, 而且场景也可以在较少的时间里被渲染。

测试一个对象是否被另一个所遮挡通常是一种复杂的运算。然而我们知道, 在这种山的情形下, 它是建立在高度场上的。由于高度场是一个关于高度的二维数组, 因此不能被任意延伸。如果一个对象在山后并且完全在由山的上部轮廓形成的地平线之下, 它就必须被剔除。

理想的情况是, 在场景里的任何山, 如果从视线上看来被在其前面的其他山所阻挡, 那么这些山应该作为对象而且应该被标记为被遮挡的。我们可以通过从前到后依次画出每一座山来实现它, 这样我们所能够看到的范围就非常明了了。

处于地平线以下的山被剔除; 剩下的则被渲染并且将它们各自的地平线轮廓跟所有被渲染山脉的地平线合并在一起。这种从前到后追踪 (track) 所有地平线的方法称为遮挡地平线 (occlusion horizon), 而这种通过测试对象是否在遮挡地平线下面来剔除对象的技术被称为地平线剔除 (horizon culling)。

本文提出了一种新颖的方法, 它通过构建地形的近似值在户外场景中实现地平线剔除, 运用这种方法可以在运行时有效的生成遮挡地平线。

地形适用性

只有场景包含了好的遮光板, 游戏才能受益于遮挡剔除。使用遮挡的一个最好的场景案例就是在第一人称视角射击游戏中, 玩家在翻越山岭时,

在户外地形上的地平面上移动，如图 5.15.1 所示。

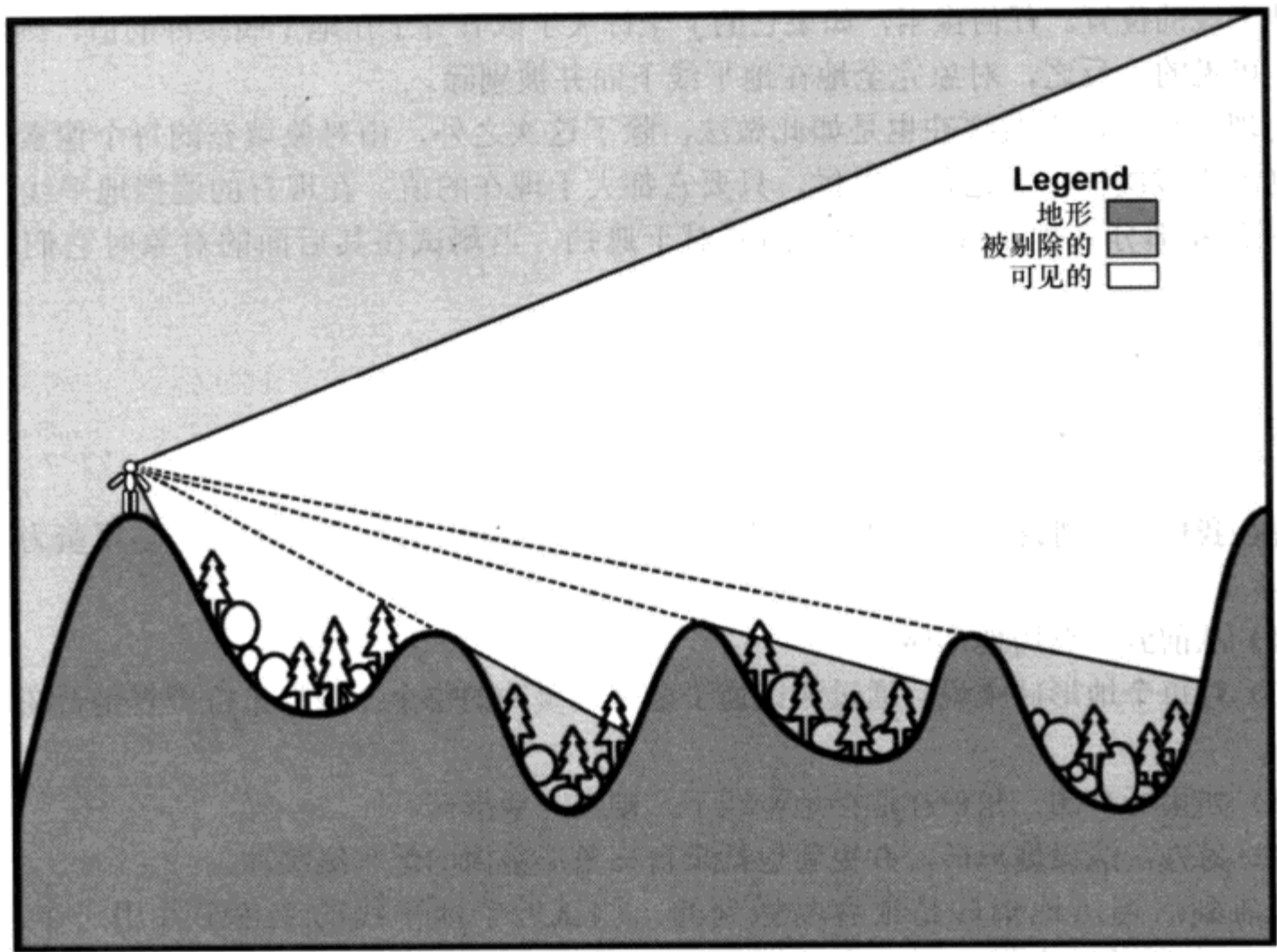


图 5.15.1 使用遮挡剔除的理想的户外场景

一个较坏的情形为地形是平的，或者为游戏中的玩家在地面上很高的位置，例如飞行模拟器。像这样的场景几乎不能或者很少从遮挡剔除受益，通常使用 LOD 技术来进行优化。大多数场景是介于这两种极端的情况之间，那么将遮挡剔除和 LOD 的组合起来使用是一个很好的选择。

在这篇文章里，我们将从建造在高度场上的四方砖构造而来的地形内容中讨论地平线剔除。为了更具体的表述，我们将假定地形按照 16×16 的砖块进行分类，而这些砖块是从高度样本为 1024×1024 的高度场上建立起来的。

5.15.2 地平线剔除基础

为了实现地平线剔除，我们将需要遵循如下关键要素：

- 从前到后遍历场景的能力；
- 一些记录遮挡地平线的方法；
- 一个测试其能够检测一个对象是否在遮挡地平线之下；
- 一个方法，其作用是将一个对象的分布掺和到遮挡地平线中。

我们将会用一个被称为地平线缓冲（horizon buffer）的屏幕空间高度数组来追踪遮挡地平线。这个缓冲是一个高度值的数组，每个条目代表地平线的高度对应于屏幕上的一系列像素。

0 值代表屏幕的底，而且正值直接映射为地平线的 y 坐标。

通过把对象所填充的像素同相应的地平线缓冲中的值相比较，就可以测试出对象相对于遮挡地平线的位置。任何像素，如果它的 y 坐标大于或者等于在地平线缓冲的值，则这个对象必是可见的。反之，对象完全地在地平线下面并被剔除。

同理，更新地平线缓冲也是如此做法，除了这次之外，由对象填充的每个像素的高度被写到地平线缓冲，无论什么时候，只要它都大于现在的值。在现有的遮挡地平线之上的对象的合并部分在地平线缓冲中，因此对于遮挡，当测试在其后面的对象时它们被考虑在内。

5.15.3 蛮力地平线剔除

现在我们有一种途径追踪遮挡地平线，实现地平线剔除的最明显的方法是用蛮力 (brute force)。

(1) 从前到后遍历地形砖。

(2) 对每个地形砖来说，通过测试逆于地平线缓冲的每个三角形来检查它是否在遮挡地平线下。

(3) 如果所有的三角形在遮挡地平线下，则剔除地形砖。

(4) 另外，渲染地形砖，并更新包括来自三角形基值的地平线缓冲。

从前到后遍历地形砖是很容易做到的。测试近于地平线的三角形并用一个三角形更新地平线是通过尖栅扫描三角形的边来实现的。这非常容易做到，因为读和写地平线缓冲具有极高的 cache 效率，而且只有在屏幕上的两点之间的高度值需要以内插值替换。

不管怎么样，我们的地形建立于一个 1024×1024 的高度场样本。由于每个高度场样本需要两个三角形，而每个三角形有三条边，总共 6 291 456 条边光栅扫描必须从前到后被执行来追踪遮挡地平线。无疑，这是不实际的，因而需要一个更好的解决方案。

5.15.4 近似值

既然用蛮力执行地平线剔除是不实际的，那么我们必须寻找一些方法来使它更有效率。一个方法是研究出高度场的近似值，能用来追踪遮挡地平线，代替直接用地形三角形。

随着一个介绍进来的近似值，地平线不再是精确的。为其包含的误差限定上下高边界非常重要。这就承认遮挡测试是谨慎的，意味着一个对象当它不是被遮挡的时候决不会被标记为被遮挡的，而无论误差在近似值里是如何的大。为了达到这一点，我们必须测试一个地形砖的上边界对在它前面的所有地形砖相对于地平线的下边界。

我们将会在一开始确立一个近似值匹配高度场，然后每次需要执行地平线剔除时重用它。如果这个近似值是有效的，那么我们就能够很容易在运行时重构地形其为测试准备的一些误差的阈值，并且写到地平线缓冲中。这就允许用在遮挡剔除中花费的时间量来权衡对应于遮挡地平线的准确度。

5.15.5 近似地平线直线

从玩家的视线沿着山项目测地平线。通过首先确立一个地平线直线的好的近似值，我们的工作将会朝向整个地形的一个近似值发展。

此地平线由一个离散集的点组成而且能够被认为是一个一维的高度场。最容易逼近它的途径是用一条地平线直线来贯穿它所包含的所有点平均高度。在这个近似值里，我们需要任何误差的上下边界，因此我们添加了穿过地平线中最高和最低点的地平线直线，如图 5.15.2 的左部分所示。

在这个近似值里会有一些误差，除非地平线是完全平面的。为了减少这种误差，我们拆分地平线成两半并且为每一半装配新的平均、最小和最大地平线直线。然后我们继续递归地拆分成两半及装配直线，直到达到基础的地平线高度场分辨率时停止。我们刚好构建了一棵二叉树，能够用来在不同程度的准确度上重构地平线。

通过在根节点开始遍历二叉树，重构地平线被执行，如果节点误差超过阈值，则递归到子节点。在一个节点中的误差数量被定义为介于它的最大和最小直线之间的垂直距离。当节点误差是可接受时，或者节点没有子节点，我们渲染节点直线并且递归停止。用这种方法的地平线重构在图 5.15.2 中被显示为不同级别的容许误差。

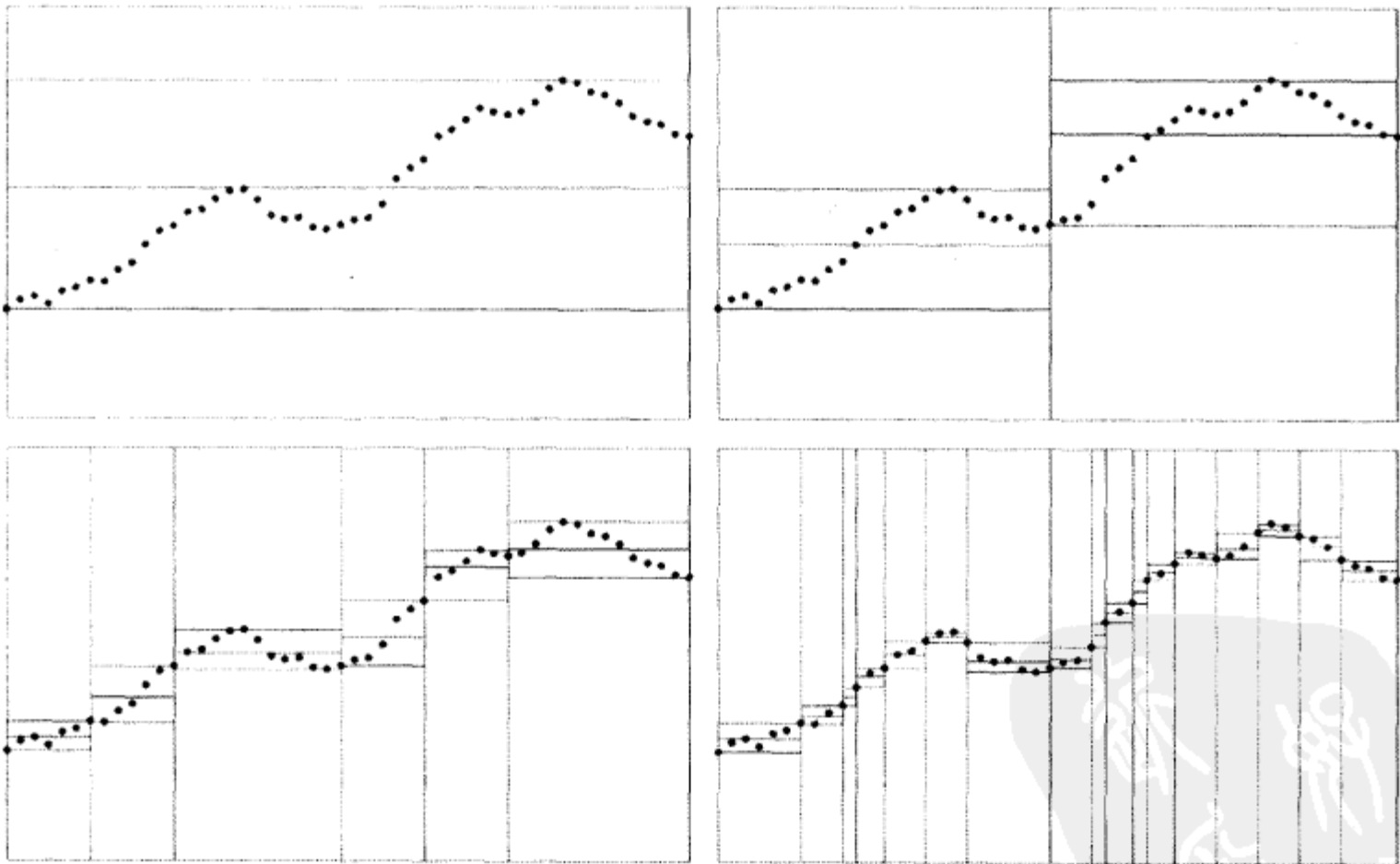


图 5.15.2 使用水平直线的地平线近似值

5.15.6 一个更好的近似值

地平线直线是很差的匹配，所以二叉树的深度递归需要得到地平线的一个有效近似值。

如果允许直线为任意倾斜，我们能够取得一个更好的近似值。

现在我们再一次逼近整个地平线，这一次用任意斜率的一条直线来代替一条水平线。我们希望这条直线是最好的可能匹配，所以误差被最小化。如果地平线总的趋向为斜向右边，那么直线的斜率应该反映这一点。正像水平直线穿过所有点的平均高度，斜线穿过所有点的平均值，被称为矩心。这确保斜率和直线的位置较好地匹配地平线的点。

如同先前一样，对任何引入的误差我们需要上下边界，所以我们添加同样斜率的两条直线穿过相关最佳匹配直线的最高和最低点。二叉树正好以相同方法确立，不过这一次我们匹配斜线来代替水平线。地平线的重构用相同误差度量递归执行的。

不同的是，这个近似值更加的准确。实际上，这意味着为了达到一个给定的误差，需要更少的递归。换句话说，为了递归深度的统一水准，在近似值里的误差合计被极大的减少了。通过比较图 5.15.3 和图 5.15.2 可以看得很清楚。

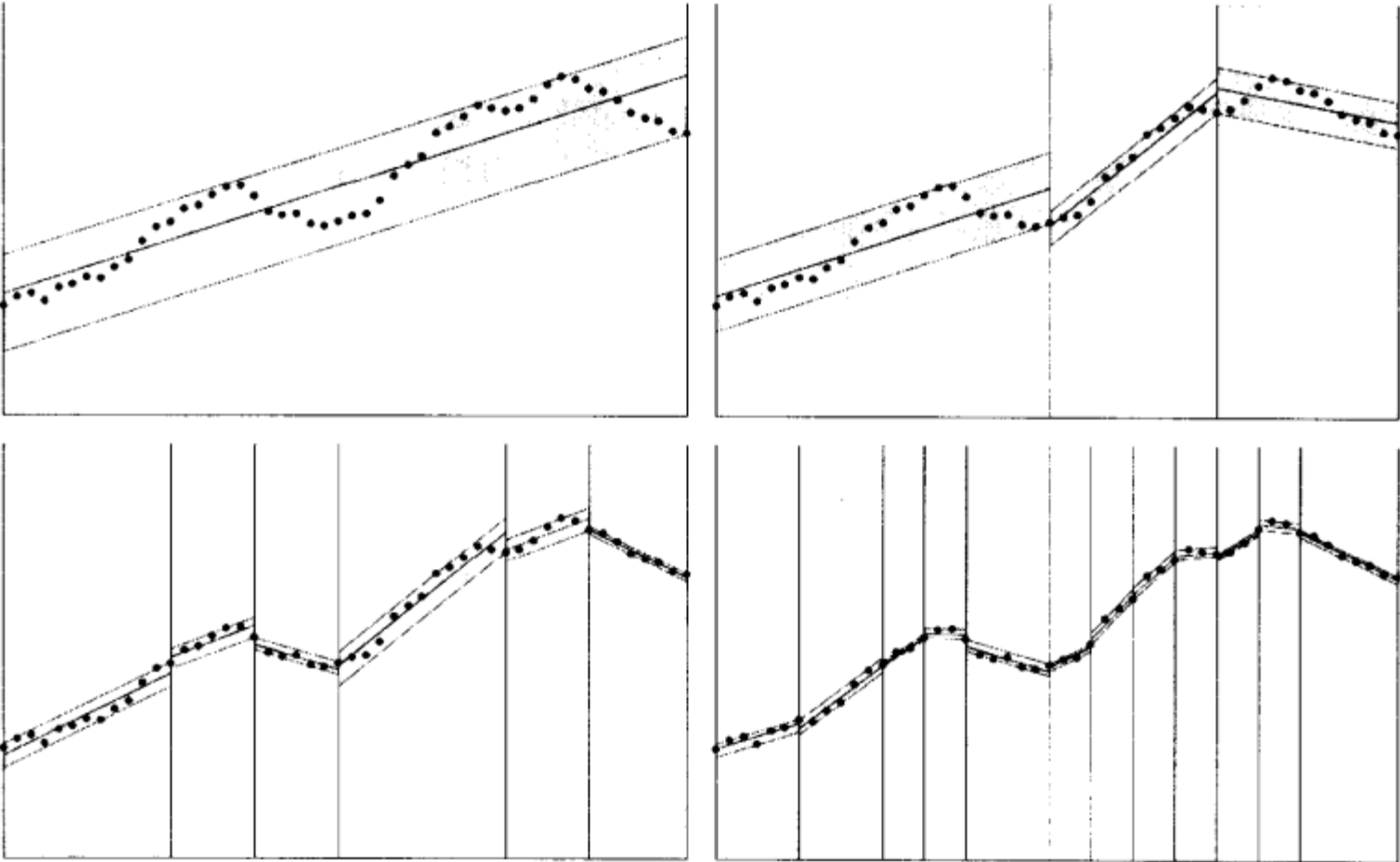


图 5.15.3 使用斜线的地平线近似值

5.15.7 最小二次方线

我们能够直观地看到什么斜率对近似直线是最佳的，但是怎样计算这种斜率让它能在代码里实现呢？

如果所有的点落在直线上，那么这个方法是无价值的。我们简单地用直线穿过所有的点。然而大多数情况下，不可能简单地找到一个完美的解决方案，所以我们需要寻找下一最佳情况，先尽可能地靠近在地平线上的点。

我们所需要做的是决定它要近似的线和点之间的误差度量，然后找到一条线使误差最小

化。出于数学上简单化的原因，最普通的方法是计算这条线来最小化介于它自己和每个点之间的垂直距离的平方和。这样的一条线被称为最小二次方线。

给出一组地平线直线中的 n 个点：

$$(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_n, y_n)$$

最小二次方线定义为

$$y = ax + b$$

$$a = \frac{n \sum_{i=1}^n x_i y_i - \left(\sum_{i=1}^n x_i \right) \left(\sum_{i=1}^n y_i \right)}{n \sum_{i=1}^n x_i^2 - \left(\sum_{i=1}^n x_i \right)^2} \quad (5.15.1)$$

和

$$b = \frac{\left(\sum_{i=1}^n y_i \right) \left(\sum_{i=1}^n x_i^2 \right) - \left(\sum_{i=1}^n x_i \right) \left(\sum_{i=1}^n x_i y_i \right)}{n \sum_{i=1}^n x_i^2 - \left(\sum_{i=1}^n x_i \right)^2} \quad (5.15.2)$$

这个结果的完整推导能够在联机的[Lauschke03]中找到。



计算最小二次方线的源代码和扫描二叉树地平线的近似值的 Java applet 包含在随书附带的光盘中。

5.15.8 将它放入到第三维中

不幸的是，我们刚才开发的地平线的近似值在实践中很少有用处。玩家的视点一移动，地平线看起来就不再是我们近似的那个了。我们真正所需要的是从任何点看都能重构地平线的方法。

如果我们应用近似值技术在整个地形高度场上来开发地平线的直线，我们能够做到这一点。因为这个高度场是二维的，二叉树变成四叉树，而且近似地平线直线部分的直线变成近似地形的正方形区域的平面。

同以前一样，当建立树时同样的概念被应用。首先，最佳匹配平面被用来为整个地形高度场而计算的，然后这个平面被上推和下推形成上界和下界。高度场这就被拆分成4块，并且同样的平面匹配处理被递归应用，直到达到高度场分辨率。

同以前一样，通过递归树直到节点，误差是可接受的，或者到达叶节点，重构被执行。这一次，节点误差度量是衍生自介于最大和最小平面的垂直距离的屏幕空间误差。近似这种误差的方法在[Ulrich02]中有介绍，应用这种误差度量的匹配一块地形的四叉树可视化如图所示 5.15.4。

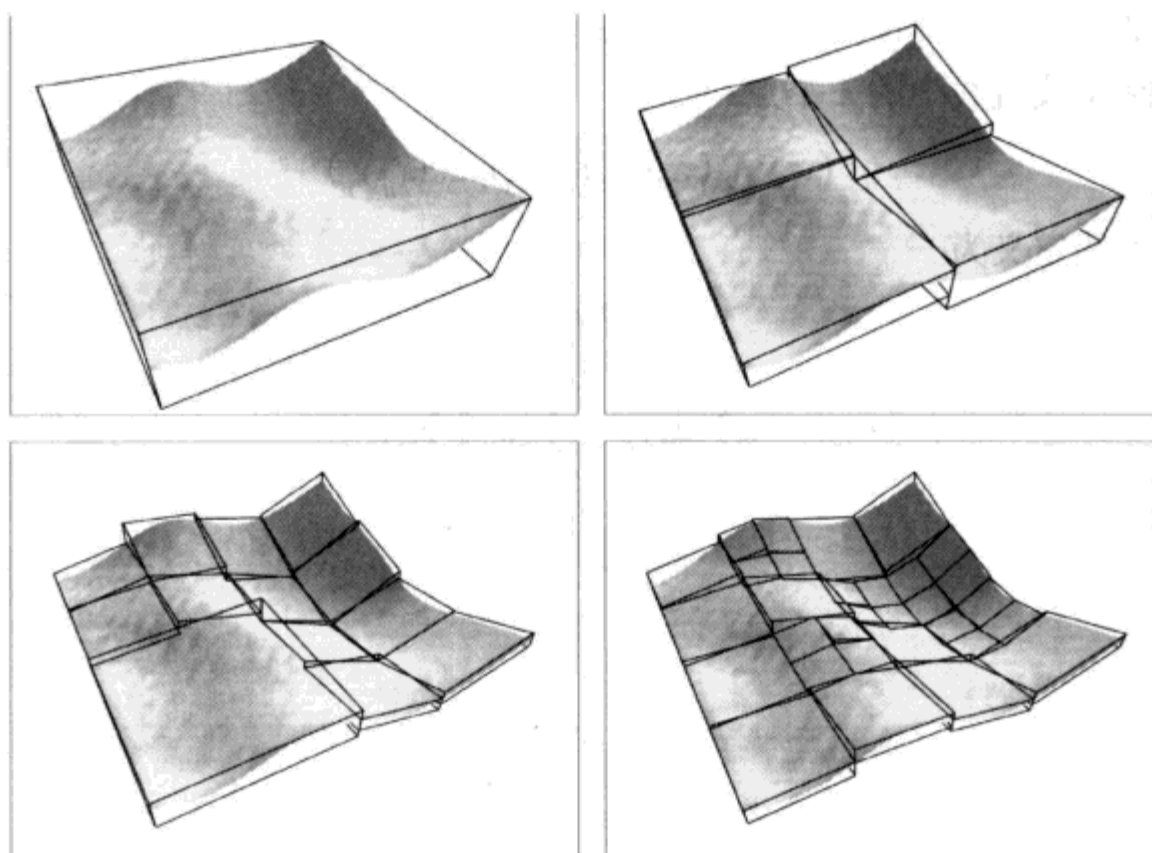


图 5.15.4 适用地平线剔除的地形近似值

5.15.9 最小二次方平面

现在我们需要匹配一个平面到我们的数据，我们必须延伸从最小二次方线到最小二次方平面。这不难做到，而且基本概念是相同的。给定高度场中的一组 n 个点

$$(x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3), \dots, (x_n, y_n, z_n) \equiv p_1, p_2, p_3, \dots, p_n$$

计算矩心 c ，为这些点的平均值。

$$c = \frac{\sum_{i=1}^n p_i}{n}$$

然后，定义一组点 q 为相对于这个矩心的高度场点：

$$(\bar{x}_1, \bar{y}_1, \bar{z}_1), (\bar{x}_2, \bar{y}_2, \bar{z}_2), (\bar{x}_3, \bar{y}_3, \bar{z}_3), \dots, (\bar{x}_n, \bar{y}_n, \bar{z}_n) \equiv q_1, q_2, q_3, \dots, q_n$$

最小二次方平面法线是沿特征向量的方向，对应前面矩阵的最小特征值。

$$q_n = p_n - c$$

$$M = \begin{vmatrix} \sum_{i=1}^n \bar{x}_i^2 & \sum_{i=1}^n \bar{x}_i \bar{y}_i & \sum_{i=1}^n \bar{x}_i \bar{z}_i \\ \sum_{i=1}^n \bar{x}_i \bar{y}_i & \sum_{i=1}^n \bar{y}_i^2 & \sum_{i=1}^n \bar{y}_i \bar{z}_i \\ \sum_{i=1}^n \bar{x}_i \bar{z}_i & \sum_{i=1}^n \bar{y}_i \bar{z}_i & \sum_{i=1}^n \bar{z}_i^2 \end{vmatrix}$$

确切结果的推导可在联机的[LSP03]中找到,代替解决特征系统的一个更简单的结果只需要一个矩阵求逆,在[Eberly01]中有介绍,而一个更加通用的匹配任意维数的最小二次方的方法能够在标准的线性代数教科书中找到,例如[Lay00]。



虽然这看起来可能使人畏惧,但请鼓足信心。这实际上不难做到,本质上只是归结为倒转 3×3 矩阵,通过所有采样点上循环计算组成的和。对于一个高度场区域计算最小二次方平面的例程提供在本书的随书光盘中。

5.15.10 用近似值的地平线剔除

我们已经开发了一个近似值,它没有深度递归并能合理地匹配地形,很容易从前到后遍历,而且有上边界和下边界。我们现在能够有效地实现地平线剔除。

取代光栅扫描三角形到地平线缓冲的是,我们用地形近似加上屏幕空间误差度量来递归到四叉树(quadtree),直到需要保证在遮挡地平线中的屏幕空间误差的那个像素。然后我们扫描四叉树节点到地平线缓冲来取代直接光栅扫描地形三角形。

节点的光栅扫描是用最小和最大平面执行的。因为节点覆盖了高度场的正方形区域,所以我们知道,节点平面横断这个正方形区域的边框形成边缘。这些边缘被以同样的方式光栅扫描到地平线缓冲作为以前的三角形边。

测试一个节点是否在遮挡地平线上是用最大平面做到的,而添加一个节点到遮挡地平线是用最小平面来做的。这就确保在近似值里的任何误差都被正确处理而且剔除测试是谨慎的。

用近似值的地平线剔除能够按如下实现。

```
function: 遮挡地形砖
  if 节点转换一个或者多个地形砖
    and 最大值完全在遮挡地平线下面
      剔除所有由节点转换的地形砖
    else if 当前节点误差是可接受的
      光栅扫描节点最小值到地平线缓冲
    else
      按从前到后的顺序递归到子节点
  end
```

```
构建地平线的四叉树近似值
main loop
  将地平线缓冲清零
  使所有地形砖可见
  遮挡地形砖
  渲染可见的地形砖
end
```

用这种方法实现地平线剔除比用蛮力方法更有益处。用地形近似值极大减少了光栅扫描到地平线缓冲边的数量,因为大多数地形大部分用平面可以较好地近似。此外,误差度量有随着距离而减少细节的性质,因为它在屏幕空间操作,需要进一步减少工作量。

最后, 近似值的分层特性意味着, 当父节点被测试及发现在遮挡地平线下时, 地形的很大一部分能够被很快地剔除。

5.15.11 被地形遮挡的对象

到现在为止, 我们仍然只是讨论了被用于遮挡测试的地形砖, 它们都被其前面的其他地形砖所遮挡。很容易扩展地平线剔除来检测对象是否在地形上, 例如被地形所遮挡的树和岩石。获得的最好结果是, 这些对象绝大多数位于地形的谷中, 如图 5.15.1 所示。

为了实现地形上对象的遮挡, 我们不得不添加它们到四叉树中。在地形砖分辨率之上的每个节点被扩展成包含一组对象指针。当一个节点在遍历四叉树时被触及, 在这个节点自己被处理之前, 它所包含的对象首先为遮挡所测试。

通过测试一个对象相对于遮挡地平线的上边界, 测试一个遮挡对象是否被遮挡就很容易做到。通过测试在对象边界球形范围之上的水平直线, 就能够被执行, 但是对象上边界的任意近似值将会做到。这条直线被光栅扫描到地平线缓冲并且用三角形和节点边来同缓冲的值相比较。如果这条线整个在地平线下, 那么对象被剔除。

5.15.12 使它成为动态的

如先前所述, 构造四叉树是从根节点开始从上到下执行的。首先, 一个最小二次方平面被用来匹配覆盖 1024×1024 高度场样本的根节点。于是, 对它 4 个孩子中的每一个, 一个最小二次方平面被匹配为它们覆盖地形的 512×512 区域, 等等。这就继续自始至终的下至被叶节点覆盖地形的 1×1 区域。

以此方法构建这样的四叉树效率极低, 因为它需要所有的 1024×1024 高度场样本必须在树中的每一级被接触。以这种方法执行, 此计算在一台典型的游戏机器上将花费几秒钟, 因此用近似值的地平线剔除怎样应用到一个动态改变的地形上?

构建树的技巧在于用自底向上代替自顶向下。当构造这棵树时, 我们首先从右边向下遍历到叶节点, 然后沿此路径后退, 当计算父节点的最小二次方平面时重用在子节点上执行的最小二次方平面计算。

这是能够做到的, 因为在最小二次方平面结果中表示的矩阵 M 只是一个合计值的矩阵。这意味着矩阵 M 的父节点不必显式计算, 它能够通过合计它子节点的 4 个 M 矩阵来创建。同样的, 父节点的矩心是子节点矩心的平均值。

现在我们能够对以它的子节点为基础的节点有效地计算最小二次方平面, 我们需要一种方法来计算沿着它的法线需要推此平面上和下多远, 如此我们能够构造最小和最大平面。这不用实际接触被节点覆盖的所有样本就能做到。因此子节点的最小和最大平面分出它们所有样本的上和下的界限, 设置父节点的平面来分出它的子节点平面, 子节点分出它所覆盖的所有抽样的界限。实际上, 这种方法比测试相对于显式的所有抽样点的方法准确度要低一些, 因为匹配一个近似值到一个近似值它累加了额外的误差, 但是它对于我们的目的是有效的。

通过重用子节点执行的计算, 我们能够有效地重建整棵四叉树, 但这仍然不够快! 为了达到需要的速度值, 我们必须只重建树的一部分, 这棵树需要更新来反映在地形中一些变化。

举例来说,如果在地形中间的爆炸过后,留下一个覆盖 128×128 高度场抽样区域的大弹坑,我们只想重新计算被受影响区域的最小二次方平面,而不是整个高度场。

如果我们首先扩展四叉树来存储矩阵 M 和所有覆盖一个或者多个地形砖的节点矩心 c ,那么我们就能够做到这一点,在前面部分被扩展的同样的节点包含一组对象的指针。现在矩阵和矩心在高阶节点被缓存,我们只需要重新计算最小二次方平面,其对于覆盖被修改地形区域的低阶节点,然后传送这些变换到树上。实际上,以这种方法传送变化到树上的工作是可忽略的,更新四叉树的花费同修改地形大小是成比例的。

这种适应地形中变化的重建能力,同其他需要大范围预处理的静态地形方法例如 [Bacik02]、[Zaugg01] 和 [Stewart98] 直接形成对比;也与需要手动放置遮光板几何学的技术例如 anti-portals (不能简单地支持动态修改) 形成对比。这是地平线剔除实现的关键,并使得这种技术在游戏中尤其适合。

5.15.13 未来的方向



ON THE CD 我们已经讨论了在一个平铺式地形的特殊上下文中的地平线剔除;无论如何,它能被应用在实际任何地形表示上,例如 Chunked LOD [Ulrich02]、Lindstrom 方法 [Lindstrom01] 和 ROAM [ROAM97]。扩展这个算法来处理地形上的对象作为遮光板是可能的。在这个方向上的一些想法可参见在 [Bacik02] 和 [Downs01] 中。硬件遮挡查询支持在此考虑上也是有用的,并且能够被加到这种技术里来更进一步地减少渲染负荷。最后,在随书附带光盘的例子程序中实现了地平线剔除,其为对于假定在视图向量上摄像机不允许有任何的滚动旋转的地形。扩展支持摄像机的地平线剔除作为一个练习留给读者。

5.15.14 结论

在这篇文章中,我们介绍了一个通过有效地剔除地平线下面的地形砖和对象来加速户外场景渲染的技术。这种技术可以被应用在运行时被修正的地形上,其更新的花费同被修正面积的大小是成比例的。

5.15.15 参考文献

[Aronson97] Aronson, "Dead Reckoning: Latency Hiding for Networked Games," Gamasutra, 1997, available online at www.gamasutra.com/features/19970919/aronson_01.htm.

[Bernier09] Bernier, "Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization," GDC Proceedings, available online at www.gdconf.com/archives/2001/bernier.doc, 2001.

[Bettner01] Bettner, Terrano, "1,500 Archers on a 28.8: Network Programming in Age of Empires and Beyond," in the 2001 GDC Proceedings, available online at www.gdconf.com/

archives/2001/terrano_1500arch.doc.

[Chandy78] Chandy, Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," IEEE Transactions on Software Engineering, SE-5(5): pp. 440–452, 1978.

[DIS94] DIS Steering Committee, "The DIS Vision: A Map to the Future of Distributed Simulation," Institute for Simulation and Training, 1994.

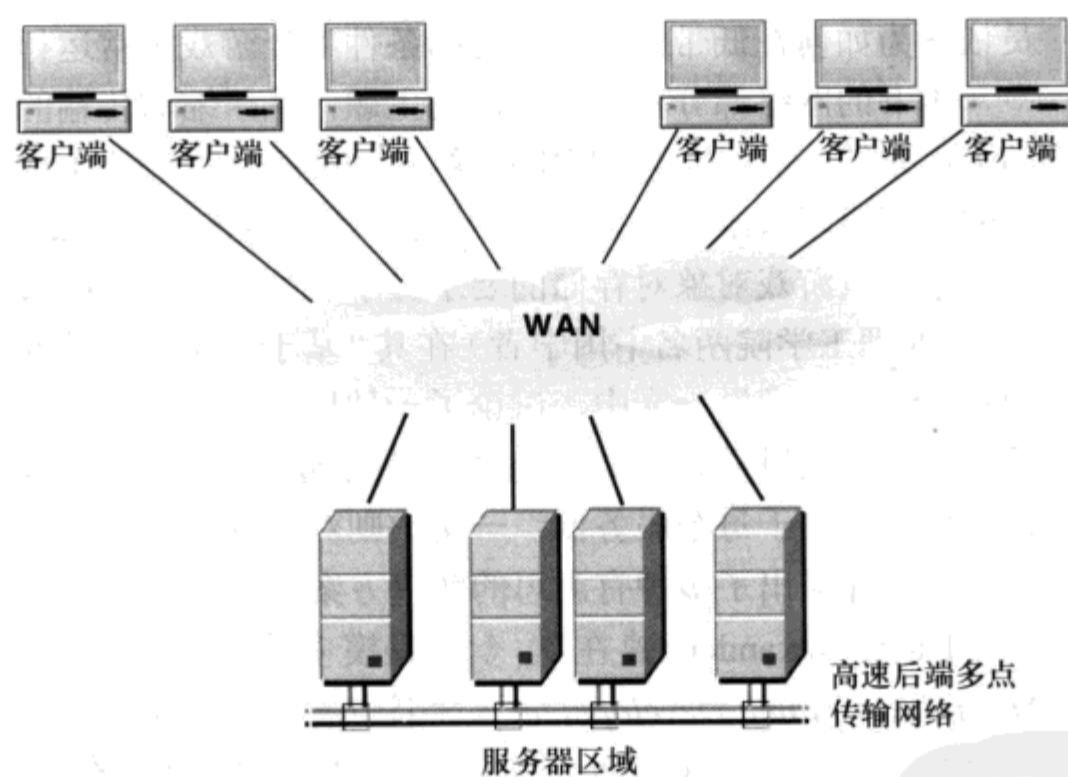
[Fujimoto98] Fujimoto, "Time Management in the High Level Architecture," Simulation, Vol. 71, No. 6, pp. 388–400, December 1998, available online at www.cc.gatech.edu/computing/pads/PAPERS/Time_mgmt_High_Level_Arch.pdf.

[Mauve02] Mauve, "How to Keep a Dead Man from Shooting," in the Proceedings of 7th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services, October 2002, available online at www.informatik.uni-mannheim.de/informatik/pi4/publications/library/Mauve2000a.pdf.

[Mills92] Mills, David L., "Network Time Protocol (version 3) Specification, Implementation and Analysis," RFC1305, 1992, available online at www.faqs.org/rfcs/rfc1305.html.



网络和多人游戏



简介

作者: Pete Isensee, Microsoft Corporation

E-mail: pkisensee@msn.com

译者: 许竹钧

审校: 沙鹰

去年(2003年),三大主要游戏机厂商(索尼、任天堂、微软)都纷纷启动了其游戏机游戏的在线业务。网络多人游戏继续其增长势头,而无线游戏亦蓄势待发,将成为把多人游戏引入大众市场的另一个媒介。尽管在这研发领域中对在线游戏专家的需求在增加,但网络还无法和它更引人注目的近亲——图形和音频相提并论。游戏网络工程师所面临的另一难点是,带宽(bandwidth)和迟延(latency)并不遵守摩尔定律。因此,年复一年,图形和AI程序员获得了越来越大的提高余地,但是网络程序员依然为如何在低带宽和高迟延的条件下让游戏正常运行而困扰。

本书的网络部分覆盖了广泛的主题,从压缩需传输的数据到设计高效的大厅系统。有三篇文章集中讨论了大规模多人游戏。Justin Quimby的“有效的大规模多人游戏状态存储”描述了游戏 *Asheron's Call* 所使用的将游戏世界中海量游戏对象对存储的要求最小化的系统。Larry Shi和Tao Zhang(佐治亚理工学院两名中国学者)在其“基于多服务器的MMORPG中的时间和一致性管理”一文中,讨论了一种技术,它能让基于服务器的游戏的反应既及时又准确,要知道这两个特征通常可是不相容的。Adam Martin在“数千用户于每个服务器”一文中则对大规模服务器研发上的很多问题作了调研并提出了多种有意思的解决方案供参考。

Thor Alexander曾在其《大规模多人网络游戏开发》(*Massively Multiplayer Game Development*)一书中展示了如何使用并行状态机来制作有效的角色动画。Jay Lee在他的文章“并行状态机在客户/服务器环境中的实际应用”中将Thor的成果发扬光大。在线游戏开发人员面临的最大问题之一是,如何制作有效的用户界面,使用户能找到好的在线会话,对此Shekhar Dhupelia在他的文章“大厅设计与开发”中提供了许多有用的技巧。最后,我的文章“位压缩:一种网络压缩技术”针对了“发送过多数据”这一许多网络游戏中的通病。

无论你是要为休闲纸牌玩家设计一个简单游戏,还是要为资深玩家设计MMORPG,也许,这些文章中有一篇有助于改进你的游戏,或缩短开发周期,或者两者兼而有之。也许,这些文章中的某一篇能增长你的见识并解决一个相关问题。无论如何,请欣赏这些作者的智慧结晶。

6.1 设计与开发游戏大厅

作者: Shekhar Dhupelia, Midway Amusement Games, LLC

E-mail: sdhupelia@midwaygames.com

译者: 许竹钧

审校: 沙鹰

在基于会话 (session) 的游戏中, 玩家之间的交流和比试均局限在某次短暂的相遇之内。这些基于会话的游戏, 包括体育游戏、第一人称视角射击游戏和纸牌游戏等, 在全世界范围内每天要被玩上数百万局。在基于会话的在线游戏的开发中, 大厅 (lobby) 是一个相当重要的方面。用户在进行游戏之前, 必须要先找到作为对手的其他玩家, 同时评估各种可用的选项。然而, 由于时间限制、计划不足或缺乏经验, 很多游戏中的大厅系统比较难于使用。可是, 若某个玩家不能轻松地找到其他玩家一起玩, 下次他多半就不愿意玩这款游戏了。

本文覆盖了基于会话游戏的大厅所需的基本功能, 重点讨论了那些既必要又有助于增加游戏重玩价值 (replay value) 的功能 (feature), 诸如积分排名 (ranking ladder)、锦标赛 (tournament) 和内容等。本文提出了一个可处理大厅所需的各种选项和失败条件的通用状态机。

6.1.1 状态—事件系统的设计

在线游戏的大厅包括几个不同但同等重要的子系统。一般来说, 单个用户在某一时间只能和某一个子系统进行交互。比如当用户正浏览聊天室列表或检查某场在线锦标赛的当前状态时, 不可能去查询哪些用户排在积分榜的前十名。

可以把在线大厅的这些子系统, 或者说模块, 想象成高层状态。这些状态就是一个当前“功能”的枚举列表, 例如:

```
typedef enum
{
    eLobbyState_Authentication,
    eLobbyState_fLadderRankings,
    eLobbyState_MatchMaking
} eLobbyState;
```

一个用户在某一时刻仅处于其中一个活动状态。这些状态可以用等级的顺序进行排列。比方说, 某个用户若是没有满足 eLobbyState_Authentication

状态的条件，就不能前进至该状态以后的任何状态。

此外，每个子系统可以有自身独立的状态枚举列表。例如，在游戏配对（matchmaking）状态中，某用户想挑战另一个用户，而同时第三个用户可能正在修改其排序标准。例如状态 `eState_InChatRoom` 可以拥有子状态 `eState_AwaitingUserList`、`eState_GameChallengePending` 等。

大厅系统的另一个重要方面就是在线事件的异步特性。虽然，追踪某个用户当前所在的位置并推测其后继动作的可能性是一个简单的过程，但是要跟踪所有的事件则完全是另一回事了，尤其是当未决的邀请或私人消息可能经过很长时间仍得不到响应时（超过了任何可能的大厅服务器负载）。

有许多在线动作需要用请求/响应的方法来设计。请求（request）是一个被发送到服务器的异步查询或事件。游戏不能停下来等待响应（response），因为若那样用户就无法改变主意、中断连接、修改其存储在本地的游戏选项等。来自服务器的响应主要由回调函数（callback function）来处理：通过执行响应代码和决定下一个合适的动作。例如，提交一个玩家名字和密码可能会产生如下的几个结果码（result code）。

```
typedef enum
{
    eLoginResponse_Success
    eLoginResponse_InvalidEntry,
    eLoginResponse_ServersDownForMaintenance
} eLoginResponse;
```

为这些用户状态增加超时检查（timeout checking）进一步提高了界面的可用性。如果某用户向另一个玩家发出挑战，而过了 10 秒之后仍没有收到响应的話，就应该马上允许该玩家发出另一个挑战，而不是无限制的等待。

6.1.2 探讨大厅的子系统

有一些服务被视为是当代在线游戏必须具备的。这些服务为用户增添了既有趣又吸引人的在线体验，通常也同时为出版商提供商业增值。在此我们对这些系统做一些简单描述。

根据之前描述的两层大厅状态架构，在美术和开发资源保持一致的假定下，这些系统可以做某种划分并由几个程序员共同开发。当开发者从大厅状态的高层俯瞰时，每个系统都能独立地由自己独特的底层状态构造而成。

1. 身份验证（authentication）

一旦用户实际在线并连上游戏服务器，他可以有两种途径来登录：匿名（anonymous）登录或使用配置文件（profile）的登录。具体采用哪种方式来实现，视乎产品的目标和所期望的用户经验而定。无论哪种情况，在用户还未通过该阶段之前，其他的大厅功能都应该是禁止访问的。

匿名系统很简单——匿名的嘛。用户键入某种用户句柄（user handle）或名字（name），然后就能访问大厅了。用户选择的字可能彼此重复，可以通过在每个重复的名字后面附加一位数字（0..9），也可以通过暗中记录某个惟一的序列号或 IP 地址来区分不同用户。匿名系

统非常简单,也无需在后台(backend)存储长期性的数据(long-term data)。

配置文件的验证需要创建一个真正的用户名和密码,可能与某些现实世界中的数据有联系,例如电子邮件地址、信用卡和邮寄地址等等。当目标是为了实现像锦标赛和排行榜(稍后会进行讨论)之类的比赛功能时,游戏就需要通过配置文件验证来惟一地跟踪每个玩家的游戏结果。

配置文件验证应该获得高度关注和大量的研究——虽然它为游戏的可玩性和商业价值提供了多种可能性,但同时要注意不同的国家和地区有着不同的隐私法。因此,在决定采用配置文件验证之前,要谨慎处理好开发过程和法律事务。

2. 配对(matchmaking)

游戏配对最常见的例子就是典型的“服务器”模型[Calica98]。在这个模型中,用户可以查看游戏服务器或主机的列表[Lincroft99]。玩家可以用诸如网络连接速度快慢、选用的游戏关卡或地图、已经连线的其他玩家的数量等与具体游戏相关的标准来对该列表进行排序。之后,玩家可以决定加入某个已经建立的游戏或创建自己的游戏。

如果排序标准对用户来说是静态的,那么最好使用文件夹系统(folder system),为可玩的游戏列表建立某种浏览顺序或者是层次结构。在体育游戏中,玩家首先要从体育馆列表中选择比赛场地,然后才能进一步从符合标准的游戏列表中做出选择。这简化了游戏的排序和浏览,也方便了游戏和后台服务器的划分。

如果游戏只在两个玩家之间进行,例如国际象棋和网球,那么可以实现一个“挑战/应战”系统。和前面所描述的主持或加入到某个游戏或服务器不同,用户只会看到其他玩家列表,可能同时附带某些惟一的数据(如地理位置、胜负记录等)。玩家向其他用户发出挑战,而被挑战者可以接受也可以拒绝。发起挑战的用户或者网络状况最好的用户都可以来主持这个游戏。

3. 聊天(chat)

在游戏中增加聊天功能从根本上方便了人与人之间的交流。现在用户不再仅仅和其他在线玩家一起打游戏,实际上,他们和游戏中的一切,甚至可能还有游戏之外的事件进行交互。聊天不但使高度动态的团队行动,连同“嘲弄”(Taunt,有身体语言之意)以及高层次的交互成为可能。为了具有市场竞争力,聊天功能一般来说被看作是不可或缺的。

类似于配对文件夹的层次结构,聊天的会话可以组织成一个一个的“房间”,这样用户的聊天状态就可以从原先的“正在聊天中”改进成“在聊天室1”、“在聊天室2”等等。然而,将聊天和配对两者分离的做法并不妥。相反,当玩家浏览其他的游戏和玩家,检索过全部的游戏排名,用聊天的形式来发出挑战和“身体语言”的效果会更好。可以将聊天加进任何形式的配对中,这样用户就可聚集在房间里,和周围的玩家畅所欲言。

与身份验证相似,提供文字或语音聊天的同时要尊重隐私和年龄方面的法律问题。要谨慎地实现聊天功能。

6.1.3 高级大厅子系统

为了增加真正意义上的重玩价值和竞争力,需要更多的大厅子系统。在一个在线游戏中,总有无穷多个尚未被实现的想法。然而,很多有竞争力或增强的功能均来自于某些核心想法。

这些想法使一些高级子系统如今成为在游戏中常见的，并且实际上正逐步归入“必备”类的子系统。

1. 内容下载

内容下载（content download）是对一系列不同功能的通用称谓，其含义包罗万象。在体育游戏中，内容下载最常见的例子是名单的更新。举足球和篮球游戏为例，在整个赛季中一般是看不到很多新球员的。但是，球队会用各种交易方式来交换球员。由于游戏中的球员模型除了正确设定着装和球衣号码之外，不需再作其他调整，所以相当容易就可以为整个体育联盟实现一个每周或每月的更新。无论是在线还是单人游戏模式中，由于球队球员名单更新使得体育迷们每次玩游戏的时候都能看到和真实的联盟保持一致的名单，增加了游戏的重玩价值。

执行下载时，先通过查询硬盘或存储卡来找到最近的更新，如果服务器端有着更高的版本号，则下载新的版本。该数据主要是简单地用符号隔开的文本数据或 XML，可能被加密或受到其他方式的保护。

内容下载的另一个范例就是新地图、新角色模型的下载。虽然这些图形附加部分必定需要大量的测试、重新提交游戏审核或者重做其他的管理工作，但它们是最受用户欢迎的下载。这在动作类游戏模拟类和第一人称视角射击类游戏中已越来越常见。这些附加能够给用户焕然一新的游戏体验。事实上，第一人称视角射击游戏的玩家在游戏原始版本发布后的几年内都会看到不断公布的新地图和其他调整，这使得该游戏摆放在零售货架上的生命周期大大延长。

新的需求伴随内容下载而来。配对界面一定要能处理新的内容，就算有玩家尚未下载新的数据，也须能让玩家做出配对决定。例如，如果玩家们按照各自喜欢的地图分成了不同的大厅房间，那么界面必须新的选项中提供新地图的下载。

近来关于可下载内容出现了其他的创意。其中一个创造性的想法就是“可下载的天气”。游戏根据玩家窗外实际的天气，在游戏中表现阴晴雨雪。地理定位通常是用玩家的邮政编码来实现的。游戏中的天气可以表示为如下的简单枚举类型：

```
typedef enum
{
    eGameWeather_Sunshine,
    eGameWeather_Rain,
    eGameWeather_Snow,
    eGameWeather_Hail
} eGameWeather;
```

在开始游戏之前应向服务器查询该值。

2. 竞争

排行榜和在线锦标赛这两个高级系统已被普遍采用。由于它们直接鼓励用户反复进行游戏以获取高分，给游戏增加了最具竞争力的重玩价值。由于这些系统在不同的游戏之间都很相似，所以将它们整合到大厅是非常容易的。

排行榜，有时也被称为“排名”，简单来说就是最好玩家的列表，换个说法，所有玩家

的列表。这可以用很多不同的方式来划分。例如，大厅的一个选项能看到所有玩家的前 100 名，而排名的方式可以是歼敌数量、所过关卡的数目、得分或解决的难题；另一选项可以是选最后 100 名，有些另类的玩家实际上在努力成为游戏的最差玩家！另一种方式就是维持一个关于所有玩家的动态列表，显示用户所在位置，同时显示排在前面和后面的玩家以便对比。这就需要每局游戏结束后将最终的游戏状态信息发送到数据库，同时在后台不断维护总排名情况。

锦标赛则稍微复杂一点——它不需要保留相同游戏状态的数据和结果，但它通常会涉及几个或一部分玩家。例如，16 个玩家可能选择在他们之间进行一场锦标赛。这场锦标赛可能有一个惟一的命名、一个登录密码、允许的玩家列表和每场比赛的结果。在一场体育锦标赛中，当玩家获胜，就会沿着锦标赛的树型结构上升，直到被淘汰出局为止。

这些功能的关键因素是费厄泼赖（fair play，公平游戏）。玩家能作弊，也愿意作弊，为了对付不正当行为，必须实现一定的逻辑规则。以下为应当用在游戏中的主要规则。

- **跟踪断线。**为了不让其他玩家获取胜利，也不希望失败的记录被写入自己的账号，有些玩家经常在最终比分递交到服务器之前退出游戏。应当把断线作为排名算法中的考虑因素之一，如胜积 1 分、失败积 0 分、断线则倒扣 1 分。这样就鼓励了游戏公平和完整地进行。

- **将用户账号和现实世界信息相关联。**另一种常见的作弊方法就是创建第二个傀儡账号（俗称大米，即 Dummy），同时登录两个账号，接着用一个账号不停地打败这个傀儡，从而人为地提高了第一个账号的排名。如果将创建用户配置文件和信用卡或 E-mail 地址这样的现实世界信息关联在一起，这种作弊的难度会增大很多，发生的几率也会大大降低（虽然这会带来很多额外的安全问题）。

6.1.4 结论

可以说游戏的在线功能，无论是比赛型的还是合作型的，在新游戏中很快会和 3D 图形一样变得必不可少。多数 PC 游戏的大作都标有在线支持的字样，这种趋势自然也可以被预知，而且相同的趋势正迅速地影响着电视游戏业 [Ganem03]。虽然游戏类型多元化，彼此在线风格亦各不相同，但各款游戏的在线大厅都有着许多共同的成分。也许新颖、独创的功能和模式会继续出现在每一款游戏中，但登录、社区、玩家配对等基本概念仍将保持不变。

设计大厅系统时，既要兼顾实现和处理的简便，又要让普通玩家方便上手、尽快熟悉，两者都很重要。当用户玩一款新的在线游戏时，应该能够又快又方便地登录并迅速找到人一起玩。这通过一种既简单一致，又具有足够灵活性可以支持那些区分游戏好坏的“高级”功能的设计来实现。为了确保能够方便地对钩子（hook）和回调函数（callback）进行维护和修改，对状态驱动设计做出流程图并在开发过程中作为参考（有必要的时候进行更新）是非常重要的。这既加快了测试和反复开发阶段，也极大地缩短了整个反馈周期。

要了解更多有关大厅设计的最优方法，就去玩游戏吧！既要玩 PC 游戏也要玩视频游戏。玩所有类型的游戏。既要试试游戏内的配对，也要试试第三方的游戏配对服务。总之最重要的是尽可能多玩不同的游戏，这样你就能汲取别人的经验，同时避免最常见的错误。

6.1.5 参考文献

[Calica98] Calica, Ben, "Multi-Player Lobbying, or, Gathering the Team," available online at www.gamasutra.com/features/game_design/rules/19980904.htm.

[Ganem03] Ganem, Steve, and Pete Isensee, "Developing Online Console Games," available online at www.gamasutra.com/features/20030328/isensee_01.shtml, March 2003.

[Lincroft99] Lincroft, Peter, "The Internet Sucks: Or, What I Learned Coding X-Wing vs. TIE Fighter," available online at www.gamasutra.com/features/19990903/lincroft_01.htm, September 1999.



6.2 支持成千上万个客户端的服务器

作者: Adam Martin, Grex Games

E-mail: gpg@grexengine.com

译者: 许竹钧

审校: 万太平



为了应付每台服务器上日益增长的巨大客户数量,在某些地方需要采用全新的技术。主要的门槛是在服务器有 2、50、500、1 000 和 50 000 个连接客户时。当客户数量有 50 000 或更多时,服务器通常要组成集群(cluster)并使用如事务处理(transaction processing) [Gray93]的方法来共同承担负载。不论是有 1 000 至 50 000 个客户连接的独立服务器,还是处于集群中的服务器,本文都是适用的。本文给出了能容纳成千上万客户规模的常规服务器的设计(配套光盘中附有源代码)。你可以使用这个服务器作为用于构建你自己的大规模服务器的起点。

6.2.1 服务器设计中的门槛

服务器开发使用了一系列的算法和体系结构,主要差别在于它们所能适应的并发客户的数量。

通常,那些处理较少客户的方法简单容易理解,也能让你立即采取行动。但是,就算在以简单版本开始的初级阶段,每种方法都是互不相同的,而且后面的升级会在实现和测试时间上付出很大代价。要是你使用了不恰当的方法,也许在你的早期测试中它运行得很完美,然而在后面的阶段将导致灾难性的崩溃。

客户: 500 ~ 1 000

异步和不中断的 I/O 系统对于 500~1 000 数量的客户会运行良好,但为每个客户创建一个线程的话,就不可能处理如此之多的连接。在如此规模的情况下仅用一个线程处理所有的客户,就算是用异步 I/O 也同样不现实。通常,这个模型中的服务器有 5~10 个线程,每个线程处理 100 个客户。

客户: 50 000+

有两种主要的方法用于客户数量超过 50 000 的服务器:一种是使用非

大众化的硬件设备，另一种则是转用一个多服务器的分布式系统——就是一个物理服务器集群，共同合作并一起分享它们中的客户。硬件上主要的限制是带宽太少不能满足服务器的不同部分，同时 OS（操作系统）处理海量客户的效率十分低下。即使每个客户使用很小的开销，合计起来它们也使用了太多负载。

除非装有特殊的硬件，Intel/AMD 的单机服务器很少接近达到 5 万连接客户的目标；举例来说，特别快的局域总线（local-bus）比如 PCI-X（133MHz 和 64 位，提供 10 倍于 PCI 的带宽[Compaq00]），或者是交叉开关总线（crossbar-switched bus），这些设备非常昂贵，但提供了极高的带宽和较低的迟延。

这些数量高度依赖于服务器对每个客户必须做多少处理，以及单个客户的典型使用模式。举个例子，如果你期望一个典型的游戏服务器能容纳大约 300~400 个玩家，你最好假设会超过 500 的门槛，然后选用恰当的服务器设计。然而，假如你知道你的服务器处理相当简单，你可能期望采用正常 50~500 玩家规模的技术，估计可以应付大约 1 000 个玩家。尽管你没有为错误留多些空间，但你可以在后来转换使用简单的分布式系统，每台服务器运行完全独立的游戏，且每台不超过 1 000 个玩家。

本文的余下部分讨论了当服务器处理 1 000~50 000 个客户时会遇到的问题及相应解决方案。

6.2.2 问题

应付成千上万个客户时，服务器会面临很多难题。理解这些问题对于成功创建这些类型的游戏服务器来说是非常必要的。

1. 非确定性

非确定性是引起最大问题的原因。非确定性意味着仅看源代码，你无法知道这段代码会做什么。一段代码确定性越少，则越难推理。当进入代码调试阶段，非确定性会产生大量不可重现的 bug，使得调试举步维艰。

大多数游戏程序设计想方设法来避免非确定性。对于避免非确定性的经典例子是大部分游戏中被使用的主游戏循环方法。这实现了游戏不同部分的静态调度，因此你总能知道每部分以何种顺序运行。然而，在要处理应付大量的客户时，不可避免地需要使用动态调度和多线程，这就立马引入了非确定性。此外，进行分布式程序设计时，每台计算机都是惟一的，像处理器速度等不同的属性则意味着代码在跨机器运行时的速率也各不相同。

2. 非平常故障模式

当客户或者服务器断线或机器瘫痪（crash）时，通常无法知道究竟发生了什么。当一台特殊的计算机停止响应时，要判定采取何种应对措施并不容易。很可能对故障假定出不正确的原因，随后采取不恰当的行动。例如，假如网络出了一个临时小故障并且破坏了玩家的会话，一些游戏就可能将这客户踢出游戏。较好的设计在面临网络故障时应该非常稳健。

3. 规模问题

规模是问题的另一个常见根源，有两个原因。第一，当你用一个很大的客户数量乘上一

个小的消耗 (overhead), 那么消耗就成了一个很严重的问题; 就算小消耗中的细微差别都可能对系统造成很大的影响。第二, 大多数游戏服务器的开发周期包括在开始阶段测试小批量的客户, 通常只是 1 或 2 个, 接着慢慢增加到 20 左右, 然后到 100, 以此类推。只有在服务器开发接近尾声时, 带有实际负载的压力测试 (stress test) 才会暴露资源管理和非确定性方面重大的 bug。

假如你有个好的游戏设计前提, 要为游戏中最大玩家数量设定一个精确的值, 那么你一定要明确地使用它, 因为它能越早和越精确地帮助你计划在测试中需要多少客户做模拟。你能够以你将永远不得不处理的最高限制做测试, 可以避免很多与过载相关的问题。

4. 过载链式反应

一旦服务器过载, 系统经常会遭受链式反应 (chain reaction), 会对游戏产生灾难后果。当服务器严重负载时, 每单元工作的处理时间就会增加。随着每个任务要花费更长的时间来处理, 窗口出现非确定行为的机会就开始增大, 以前没出现过的全新 bug 也会出现。最麻烦的是当系统负载不高的时候, 这些 bug 一般不会重复出现。

同时, 可用资源 (CPU、RAM 等) 的减少降低了同一台服务器上其他进程的吞吐量, 因此它们也开始变得过载。解决第一个过程的过载可能只需要很短的时间, 但该链式反应带来的副作用可能会消耗系统大量的时间。这就是为什么你需要谨慎额外防护 (over-provision) 你的系统, 因此在“正常情况”下, 服务器工作比正常要求还要快些。这就保证了, 当引起暂时过载的因素消失时, 服务器能迅速返回到较为轻松的负载状态。

5. 人为因素和恶性循环

玩家对你游戏的影响是很难预计的, 而且因为有成千上万个玩家, 他们可能对你的系统造成巨大的影响。不但玩家不可预知, 而且有时候他们将以最坏的可能方式来给游戏服务器带来问题。举个例子, 当一台服务器遭到一个暂时的麻烦且无法处理时, 玩家会不停地点击按钮, 试图消除因为游戏的无反应而带来的失望。这就导致引入请求和命令数目暂时增加。

假如一台服务器需要很长时间才能从暂时性的过载中恢复, 那么恶性循环就可能导致崩溃。这个问题发生的通常方式是一个过程将资源消耗光。关于此的一个例子是, 黑客已经用有名的异步攻击 (SYN-flood) 方式来让 Web 服务器崩溃: “在某些时候, 系统可能耗尽内存, 崩溃, 或者是以其他形式表现出无法工作” [CERT96]。

6. 公平调度

由于服务器有成千上万的客户, 当在决定下一个处理什么时, 你不能简单地在所有客户中重复甄选。在一个典型的游戏中, 每个客户向服务器提交新命令的速率和从服务器返回的响应速率是成比例的。假设每个响应要花 10 毫秒, 那么当有 50 个客户时, 响应时间的范围在 10~500 毫秒之间 (假设所有的指令同时发出)。相对于正常情况下玩家的决策时间, 这是足够短的。然而, 当有 10 000 个连接客户时, 并且是用一个单线程按顺序服务它们的话, 范

围就会在 10~100 000 毫秒之间。就算响应时间不到 1 毫秒，该范围也仍在 1 毫秒到几秒钟之间。恰好在客户和服务端之间有额外的网络延迟，这个范围已足够宽得让一些客户无法得到处理。

6.2.3 主要技术

有一些普通的方法来处理对于任意一个大型在线游戏而言都很平常的扩充性 (scalability) 问题。在本节中，我们分析当客户数量达到成千上万时的性能挑战，也讨论了这种大型系统所固有的测试和可靠性问题。

1. 资源分配

因为有成千上万连接的客户，跟踪对于每个客户正预留多少资源 (CPU、内存、数据库连接等) 是非常重要的 (见图 6.2.1 所示)。三个主要选择如下。

- 普通分配：为 N 个客户创建 N 个资源；不存在资源共享。
- 存储池分配：为 N 个客户创建 P 个资源， $P < N$ ；资源可以共享，或者基于先来先服务的原理进行分配。
- 分阶段/管道分配：管道分为 S 个阶段，有 S 个资源。S 独立于 N，即客户数量。这提供完全可预期的性能和行为。
- 批处理分配：类似于共享分配，在任何给定时间段 (请求仅能用流逝的时间来衡量) $B < R$ ，为 R 个请求分配 B 个资源。

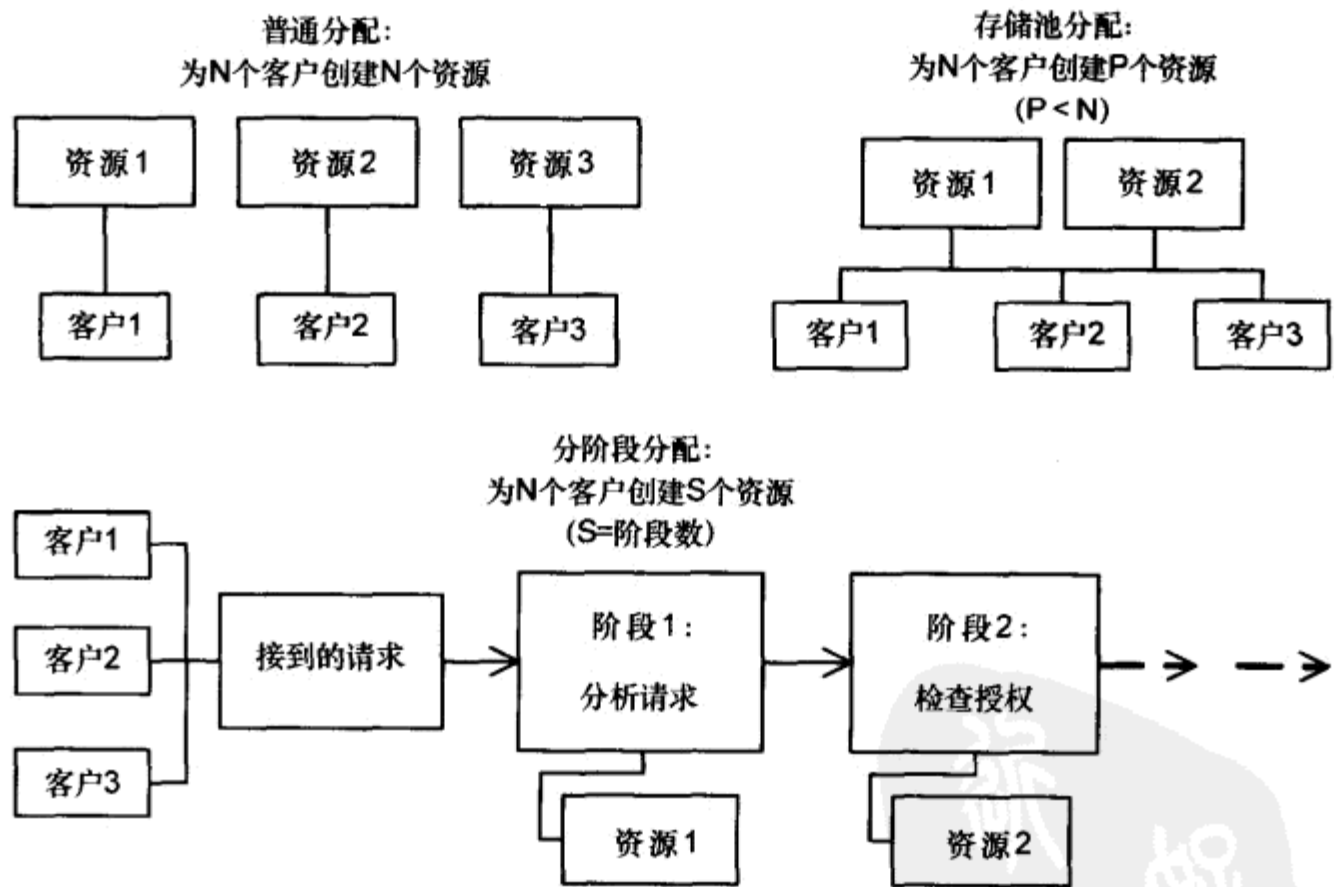


图 6.2.1 不同的资源分配策略

普通分配实现起来非常简单，但只适用于客户数量为几百左右的规模。最常见的例子就是对于即将进入的连接按照一客户一线程 (thread-per-client) 分配资源。存储池分配使用广

泛，以几种不同的模式出现。最高级的模式包括对线程池做负载平衡 (load balancing)，其中有一个额外的线程专门用来分配池中的稀有资源。该管理线程尽量使其他线程的工作吞吐量达到最优，这类似于很多 OS 的线程调度程序[Josephs03]。

分阶段分配最容易扩充，同时也很容易用于分布式系统中。每个阶段彼此独立并且不需要重新编码就能运行在独立的物理服务器上。由于每个阶段能够被独立地测试，所以也很容易在该类型系统中找到 bug。最流行的能支持几千个客户的 MMOG 服务器系统都使用了不同模式的分段。如果你想要的设计超出了本文所描述的服务器规模，那么你最好借鉴一下 SEDA[Welsh03]所介绍的分段作为起点。

批处理分配主要设计用于存在与启动相关的迟延且与工作量无关的情况。典型例子就是访问数据库，执行 1000 个请求来获取表中每一行的速度可能远远慢于执行一个请求来一次获取所有的 1000 个记录。数据库的连接通常是缓冲池共享并重用的，这样就可以避免重新连接、谈判 (negotiating)、登录等产生的消耗。大多数服务，包括一些数据库，会自动执行内部批处理；但那些不这么做的，使用批处理分配则能使速度提高几个数量级。对于那些不是为了游戏服务器（就是说，每秒有几千个小请求，而不是每几分钟一些大请求）模式设计的第三方产品，这个通常特别有必要。

2. 处理非确定性

在理想的世界中，所有代码将是自动保护的，每种方法都不加区别地做 assert 操作，遭破坏的数据的蔓延能够自动被发觉。但在现实世界里，软件只要“能够胜任就行”。大多数程序员所使用的 assertion，恰恰是契约式设计 (design-by-contract) 的最弱方式，换一种强一些的如[jContractor98]中讨论的契约，效果会更加好些。但这些需要付出更多的努力来维护，而且在标准应用中从仅仅几个问题中获得的收益是很小的，所以这就成了可有可无的方法。当一台服务器有很多要不间断运行几个月的交互程序，情况就截然不同。随着时间飞逝，几乎每小块代码都和其他小块进行着交互，因此，哪怕稍微使用一些契约（或判断提示）都在损害控制中非常有效。

在开发时间非常宝贵的地方，最好能少一些检查，让每个检查测试更多复杂的行为和常量，因为这能够测试到不同进程之间更细微的交互。分阶段资源分配中，在每阶段开始都需要有些检验，这是为了从一开始就防止损害数据，从而避免扩散到整个系统。

非确定性问题在服务器中频繁发生，部分原因是，大多数服务器一次要运行几个星期甚至几个月。要尽可能减少这些错误的有效途径就是经常，最好是自动地重启游戏服务器。重启做得越完善，则越好——除了游戏数据，不应该有任何状态保留在服务器上。虽然这个方法很笨拙，但若游戏设计许可，它是个非常有效的解决方案。

3. 不同类型的服务器

游戏程序设计中有三种类型的游戏服务器。

- 请求-响应 (request-response)，每个请求收到一个响应，最重要的数据是响应的内容。
- 命令流 (command stream)，客户仅发出游戏命令流。服务器对收到每条命令可能确认，也可能也不作响应。

- **订阅通道 (subscription channel)**, 客户连接除仅仅发送基本的握手 (Handshaking) 信息之外不发送其他信息。客户会收到服务器发出的信息流。常见例子包括聊天信息或者游戏状态。



ON THE CD

所有类型都有各自不同的性能特征。实现类型二的服务器一般没什么问题, 但类型一和三则可能表现出严重的性能下降, 这通常表现在既要发出大量的数据又要为每个请求 (类型一) 或发出的消息 (类型三) 做一定处理的时候。配套光盘中服务器实例就是类型一, 结合了类型三作服务器初始化消息 (需要发送给客户的消息, 但它不是对任何特别请求的响应)。最可能的情况是用两台类型一的服务器来代替, 一台在服务器端, 另一台在客户端, 但这会使得客户端逻辑复杂很多。也可能只用一台类型一的服务器, 让客户不停地轮询更新, 但这会无谓地浪费大量性能。

4. 自动化测试

在开发《模拟人生在线》(The Sims Online, TSO) 的时候, Maxis 就为非确定相关的严峻问题所困扰[Mellon03]。他们使用了很多技术来克服这些问题, 其中最有效的就是自动化测试。有两个主要方法来执行自动化测试:

- 一再地重复一个短测试, 在连续的测试中间完全 reset;
- 长时间的运行一个测试, 让它完全“浸透”。

频繁重复的短测试对于发现那些仅半重复 (semi-repeatable) 的 bug 特别有效。长期运行的测试能发现那些经过很多小时或者几天连续运行才会出现的 bug。

由于测试自动运行, 所以它们可以在无人看管的情况下一次运行几天。它们也能很好地用来测试当每个单元加大工作量时的运行状况, 和应该被重用作为压力测试的组成部分。这 and 传统压力测试的主要不同在于, 自动化测试在项目的一开始就要完成, 而传统测试通常在开发的最后阶段使用。在某些情况, 不到服务器基本完成, 进行压力测试没有意义, 但更简单的自动测试则不同。对于大规模服务器, 该类型的测试能发现很多的 bug, 所以你需要在开发的早期就开始着手。

TSO 项目组发现, 就算是非常简单的测试都能很快见效。简单的测试可以组合起来 (这反映了你服务器的工作方式), 产生更复杂的测试。当一个复杂测试暴露了非确定性问题时, 将其分解为原来的简单测试, 对每一个进行测试来隔离问题。如果这样不行的话, 集中全力检查这些简单测试的接口, 因为这很可能就是非确定性开始的地方。

5. 异步 I/O

异步 I/O (AIO) 使得任意多的线程能服务任意多的连接。没有 AIO, 每个客户需要一个线程。要提供高效的 AIO, OS 必须提供对低级数据结构和 I/O 原语 (primitive) 的访问。不幸地, 这会暴露很多 OS 的内核, 同时不同的 OS 处理 I/O 的方式各不相同。目前主要有 4 种可用的 API [Kegel03] 来作异步 I/O。虽然许多已被移植到其他平台, 但还有不少 (包括 Windows 完成端口) 还没被移植。



配套光盘中的源代码实例用 Java 编写,这里使用的名字也来自 Java。Java 1.4 版介绍了一种常见的 AIO 的 API,是由 Sun 推出的,由于为本地的异步 API 使用了相当有效的包装器 (Wrapper),所以可运行在 Windows、Solaris 和 Linux 平台上。这个 API 类似于很多主流的异步 API,并且同时借鉴了 Windows 完成端口 (completion port) 和 Unix AIO (Asynchronous I/O, 异步 I/O) 技术,所以看起来是最新的一种进化版本。然而,不要只是因为它们共享同样的术语,就假定你的 API 和那个 Java API 有着相同的效率——看一下你的 API 文档。

Java AIO API (叫做 “NIO”) 中三个关键元素是如下。

- **SelectableChannel**: 一个 I/O 通道或管道。
- **Selector**: 自动管理 I/O 通道组。
- **SelectionKey**: 控制每对 Selector 和 SelectableChannel 关系的信息。

[Sun-nio03]、[Sun-channels03]、[Hitchens03]中有很多关于使用这些原语细节的解释,然而,将它们在服务器中结合的最有效的技术很少有涉及。

在分阶段的类型一服务器,像接受连接,读请求和写回答等每个主要的功能有单独的 Selector 是非常有用的。这降低了功能间的耦合,从而每个都能独立地开发、测试和调整。实例服务器使用了这个策略,并为每个不同的“系统”(如物理系统或聊天系统)分配了一个阶段,其中,系统每接收一个请求则产生一个回答。

在标准情况下,你从进来的字节流 (ByteBuffer) 中读入数据直至得到一个完整的请求,再做些处理来产生响应,然后将响应写入到另一个字节流,用一个 SocketChannel 来发送内容。由于字节流分配和回收的消耗比大多数 Java 对象要大,因此尽可能多地重用字节流是很重要的。



如果你的服务器产生的典型响应长度上相差较大,或者有很长或很多相同的节,那么建议你最好用集中写的方式。这就允许 SocketChannel 在发送一个响应时将一组字节流结合起来。为了避免过多的数据拷贝,该结合可以用低级 OS 的例行程序来完成。这也使你能够预生成部分响应,在需要的时候可以重用,每次你要重用一串字符串时也无需拷贝内容。可惜地是,在一些平台上的当前实现还存在一些 bug (具体细节见配套光盘)。

6. 处理过载 (Handling Overload)

无论服务器的实现有多好,它都可能在某点开始过载。最重要的是在这种情况下决定做什么,然后记住用这个策略来编写你服务器的各种组件。玩家的数量达到峰值,或者内部的迟延造成请求/响应聚成一团,都可能导致服务器负载达到峰值。最差情况的设定是没有处理过载。当过载发生,游戏会急剧下降到一个无法玩的状态。试图连接游戏的新玩家会发现游戏“中止”。当客户事实上已经长时间等待服务器来处理时,他们会假定它已经崩溃。另外,每个坚持等待进入的玩家会造成游戏更加慢。

前两个步骤先是停止负载的进一步增加,然后开始降低负载。这除了提高你服务器的性能以外,使得过载也会很少发生。有些服务器颠倒了这两步的次序,草率地断掉客户的连接,

这仅仅使那些客户马上打算去重新连接而已。

对于试图使用过载服务器的新客户，需要指定为最高优先级。目的是为了让它们尽可能快地离开，也避免它们很快地返回。最好发送一条特殊的消息，使得客户在一段给定的时间内拒绝重试。这段放弃时间应该是随机产生的，否则当大量的客户在同一时间全都试着重新连接时，以新形式表现的过载可能会发生，就会导致服务器在不过载和过载之间振荡。

降低现有的负载可以通过游戏中决策（退出过分活跃的玩家），或者是在提高吞吐量算法中用一个变量（将在下节给出）来完成。断开由那些算法评估得到的 10% 最不合意的客户，这一做法可以在极端情况下使用。过载的服务器通常对谁都没好处，在大多数情况下，有部分断线的玩家好过游戏一点也不能玩。

7. 提高吞吐量：最短剩余处理时间（SRPT）和最快连接优先（FCF）

最短剩余处理时间（SRPT）和最快连接优先（FCF）是从传统 OS 调度程序演变而来的调度算法，现在被广泛应用于 HTTP 服务器中。它们特别适合响应包含最多数据（类型一和三）的服务器。

使用 SRPT，每个候选都需要被检验（或一个明显的请求，或一个未决的回答），并给定一个处理等级。较低的排名意味着执行任务需要较少的处理。举个例子，如果响应在缓存中，给它+0。不在缓存中的响应则得到+2。如果请求包含了复杂的计算，它会得到+3。然后，所有的候选者被排序，排名最低的最先处理。与直感相反，饥饿不一定是个问题[Bansal01]，只要候选者被拒绝就降低它的等级是明智的，这样可以确保所有的候选者在合理的时间内得到处理。

通过最先处理最小迟延连接可以缩短外出消息的响应时间[FCF02]，FCF 就是根据这个观察结果得到的。这通常用协议或 OS 特定的知识（如内核）来完成。然而，大多数类型三服务器能够很容易地通过软件实现。通常，这些服务器发送同样的数据给很多客户；而不是为每个连接的客户维护一个“发送字节”缓冲器，服务器为接收到相同响应的每个组维护一个单独的队列，每个客户维护一个指针，显示在那个队列中它已经传送到哪个位置。最快的连接将是指针在缓冲器中位置最深的那个。在一个典型的 Buffer 类中的方法 `duplicate()` 能很好地支持这种方案，因为它保存了内存空间，但为每个复本维护单独的“当前位置”的指针。

8. 应用级缓存

客户缓存编程(client cache programming)是有关如何最大限度地利用已有的硬件缓存的，但在网络/服务器（“应用程序级”）高速缓冲器中，你必须用软件来实现缓存。高速缓冲器通常是提高性能的最有效方法，但由于它是非自动化的，你不得不实际决定来实现它。虽然表面上看，高速缓冲器对你的游戏毫无价值，但哪怕只对一些数据缓存一秒钟也能明显减少 CPU 的负载。在有些地方高速缓冲器看起来是不可能的，这是因为没有两个请求会产生同样的响应，有时候值得把请求分解成几个子请求，然后这些请求中的部分就能够被存储在高速缓冲器中。这个过程需要用 CPU 瓶颈的知识和基准来控制执行。类似其他的优化模式，高速缓冲器直到主服务器是完成的状态后才应该被实现。如果你实现了对周围的类是透明的缓存，那么很容易在后面阶段将它添加进去。

6.2.4 服务器设计

图 6.2.2 是一个 UML 的状态图，它显示了服务器的 4 个阶段。每栏表示了一个特定的类中正在发生的活动。

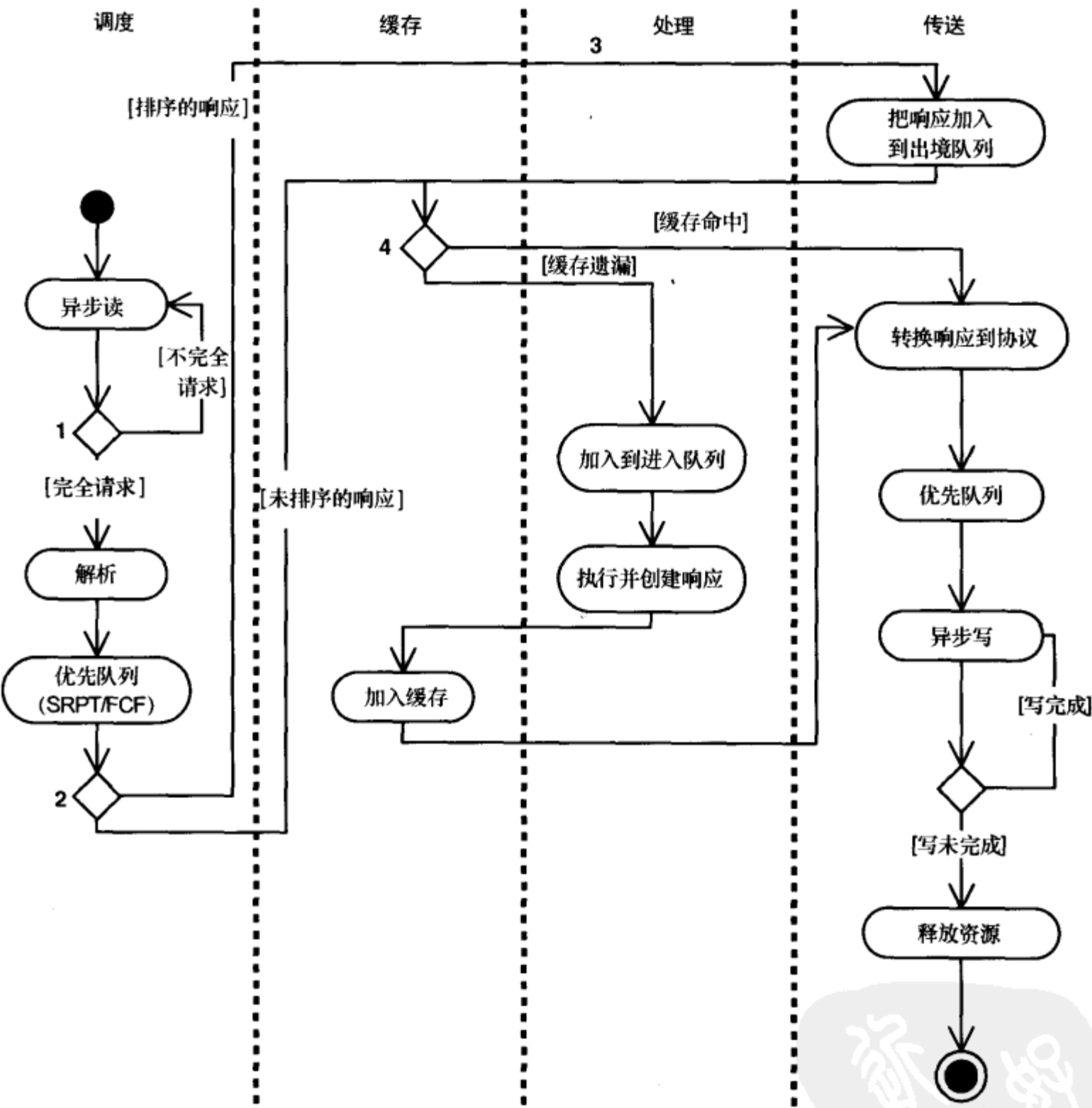


图 6.2.2 服务器设计概括

重点

- 预解析阶段 (pre-parse stage) 需要能迅速判断 Buffer 是否包含一个完整的请求。虽然这可以用试图解析和在首次失败时停止的方法来解决，但在一个异步服务器里，这样的话，为单个请求就可能重复解析很多次。这就为在客户—服务器协议中有详细的 MSG_START

和 MSG_END 字符串给出了很好的理由。

- 一些协议, 如 HTTP[RFC2616], 需要单个用户发出一连串请求, 然后响应按照所发请求的相同次序到达。在 HTTP 中, 这很必要, 因为没有其他方式可以知道哪条响应和哪条请求相匹配。对这些协议而言, 服务器必须在处理请求之前创建一个“响应对象”, 再把那响应对象插入到出境队列 (outgoing queue), 以此来保证有次序的响应。弄错的话会产生难以追踪的 bug。

- 处理 (process) 栏表示了你可能拥有的众多处理阶段中的一个。主要例子包括有一个用来移动、世界观察和配置的处理阶段。

- 通过缓存, 命令被转移到适当的阶段, 这是用一个简单的散列表将“命令名”映射到“处理对应命令的阶段”来实现的。

6.2.5 结论

海量客户的服务器开发困难且无法预计, 也可能因为疏忽大意而遍地陷阱。非确定性的很多原因会导致服务器在所有测试中看起来工作地近乎完美, 但在上线部署时就会死得很惨。有时候最有效的技术是那些缺乏技巧的 (如每 24 小时重启服务器), 但就算是这些都会大大地减少受挫。对其他部分而言, 对特别的游戏有必要甄选和整合技术, 同时要注意观察大一些的场景: 当有几千个客户时每个的工作表现, 还有在非常不确定的环境中它的脆弱程度。

与其试着一个个解决难题, 不如用本文中的技术打好基础, 再向前迈进。你也可以用这些知识, 发挥想象, 看怎样才能最好地扩展或改善你的服务器。

其他的引用和纠正可见[Martin03]。

特别感谢 D. Blake、G. van den Driessche、J. Fowlston 和 J.C. Lawrence 对本文的撰写所起到的帮助。

6.2.6 参考文献

[CERT96] CERT, “CERT Alert CA-96.21,” available online at www.mycert.org.my/network-abuse/dos.htm.

[Compaq00] Compaq/HP, “PCI-X Frequently Asked Questions,” available online at <http://h18000.www1.hp.com/products/servers/technology/pci-x-qa.html>.

[Eiffel03] Eiffel Software Inc., “An Introduction to Design by Contract,” available online at <http://archive.eiffel.com/doc/manuals/technology/contract/>, 2003.

[FCF02] Murta, C., and T. Corlassoli, “Fastest Connection First: A New Scheduling Policy for Web Servers,” available online at www2002.org/CDROM/poster/110/, 2002.

[Gray93] Gray, Jim, and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishers, 1993.

[Hitchens03] Hitchens, Ron, *Java NIO*, O'Reilly, August 2002.

[jContractor98] Karaorman, Holzle, Bruno, “jContractor: a Reflective Java Library to Support Design by Contract,” available online at www.cs.ucsb.edu/labs/oocsb/papers/TRCS98-31.pdf,

December 1998.

[Josephs03] Josephs, Mark, "Scheduling Algorithms," available online at www.scism.sbu.ac.uk/ccsv/josephmb/CS-L2-OS/oss/week9.html#Scheduling Algorithms, June 2003.

[Kegel03] Kegel, Dan, "The C10K Problem," available online at www.kegel.com/c10k.html, June 2003.

[Martin03] Martin, Adam, "Supporting Material for GPG4," available online at www.grexengine.com/sections/people/adam/gpg4/, July 2003.

[Mellon03] Mellon, Larry, "Automated Testing of Massively Multiplayer Games," *GDC 2003*, available online at www.gdconf.com/archives/2003/Mellon_Larry-AutomatedTesting.ppt, March 2003.

[RFC2616] Fielding, R., et al., "Hypertext Transfer Protocol—HTTP/1.1," available online at www.w3.org/Protocols/rfc2616/rfc2616.html, November 7, 2002.

[Spacewire03] Spacewire, "Spacewire Crossbar Switch," available online at www.mrcmicroe.com/SpaCroSwit.htm.

[Sun-channels03] Sun Microsystems, "java.nio.channels: Java 2 SDK SE Developer Documentation," available online at <http://java.sun.com/j2se/1.4.2/docs/api/java/nio/channels/package-summary.html>.

[Sun-nio03] Sun Microsystems, "java.nio: Java 2 SDK SE Developer Documentation," available online at <http://java.sun.com/j2se/1.4.2/docs/api/java/nio/package-summary.html>.

[Welsh03] Welsh, Matt, "SEDA: An Architecture for Highly Concurrent Server Applications," available online at www.eecs.harvard.edu/~mdw/proj/seda/, March 2003.



6.3 大型多人游戏状态的有效存储

作者: Justine Quimby, Turbine Entertainment Software

E-mail: Justin@TurbineGames.com

译者: 许竹钧

审校: 万太平

大型多人 (MMP, Massively MultiPlayer) 游戏是数千人可以在同一虚拟世界中居住和冒险的在线游戏。MMP 有一个独特的技术障碍需要克服——大规模的扩展性。服务器代码必须能应付数千玩家发现、拼杀、收集、使用并且触发数千游戏对象, 并且每个对象拥有许多游戏服务器使用的变量。正是这些数值对运行时内存的使用和游戏状态大小都造成扩展性威胁。本文引入了一种编程模式来解决游戏状态臃肿的问题。

6.3.1 MMP 的问题

每个游戏对象都相对于游戏世界里面其他对象有惟一的状态, 比如说巨型蜘蛛的生命值, 一把涡轮射线枪 (railgun) 剩下的子弹, 或者一名酒吧招待在最后 5 分钟内卖掉劣质啤酒的次数。尽管对象之间存在巨大差异, 但很多对象都有从一个装备 (item) 到另外一个装备很少改变的状态。举例说, 所有的魔兽 (orcs) 都有携带最多 4 件装备的能力, 所有的门都会在 14 秒之后自动关闭。最简单和最少内存的有效实现将存储每个对象每份拷贝的全部数据集。这条途径适用于少量的数据集合, 但当对象数量急剧增加时, 它就开始瘫痪了。

在极端情况下, 想象一个相同世界空间里面支持 100 万个用户的游戏。假设每个玩家有 100 件装备的物品栏, 在那里每件装备需要 1kB 的内存需求量, 那么该服务器主机群就算只是为了追踪玩家的装备清单就需要 100GB 的磁盘空间。从运行游戏服务的角度来看, 这就意味着数据备份时间延长, 数据出错的风险上升, 还有在游戏状态的回滚期间服务器停机时间也延长。

对于游戏状态原始存储的其他缺点则和运行时内存使用相关。在上述的百万级玩家游戏中, 10 000 个并发的玩家就能迫使服务器主机群要提供 10GB 的 RAM, 而这仅为了追踪玩家物品清单的需要。很明显, 这是站不住脚的情况。

就和很多计算机科学中的问题一样, 它值得偷点懒而无需多此一举。这些 MMP 游戏扩展性问题重复了操作系统设计师数十年来已经与之斗争

的问题。在多线程操作系统中，有些情况下一个程序需要复制（fork），创建与该线程一个独立的拷贝。一个单纯的复制实现只不过为每个复制的过程创建了地址空间的新拷贝。然而，拷贝一个地址空间是个昂贵的任务。有时候，复制的过程实际不会修改地址空间，意味子过程可以和父过程共享地址空间，从而节省了拷贝的花费。

现代的操作系统对于这种情况做了优化，当新的过程需要它的地址空间时通过使复制过程仅仅产生惟一的地址空间。这个技术被称之为“根据写入的数据进行复制(copy-on-write)”。一种更加正规的 copy-on-write 操作是“一种技术，在复制初始化的瞬间通过复制仅仅修改的数据，用于维护一组数据即时复制（point-in-time）操作。最初的源数据用于满足对于源数据和及时复制未做修改的读请求” [SNIA03] 。

根据写入的数据进行复制提供一种方法，减轻了对于有非常巨大数量数据集的游戏中运行时内存使用和游戏状态存储方面的扩展性问题。Qualities 对是一种根据写入的数据进行复制数据结构，它封装了对于对象游戏状态的存储方面的根据写入的数据进行复制机制。

6.3.2 Qualities 理论

Qualities 是 Turbine 公司设计的用于在关键字的值对中存储对象游戏数据的机制。整型、浮点数和布尔型都是 Qualities 能够存储的简单数据类型。对于每一种类型，每个数据的值通过惟一的关键字来引用。这个关键字用来索引 Qualities 中与它相关联的值。图 6.3.1 中给出了一些例子。复杂的数据类型也能够存入到 Qualities 中，但因此产生的问题超出了本文的范围。

索引关键字能够通过多种方法来分配。只要在关键字的命名空间（namespace）没有冲突，关键字是如何被赋值的实现则无关紧要。消除命名空间的冲突达到通过一哈希查找表以 $O(1)$ 的时间访问任何指定数据成员的能力。Turbine 引擎使用的实现是通过 typedef 定义对于被存储的每个值的关键字。不利方面是，对于想使用 Qualities API 的每个类都必须包含包含（Included）typedef 的这个列表。结果，每次增加一个新的关键字时，每个#include 头文件的文件就需要被重新编译。按照逻辑组团（logical grouping）把 typedef 分成单独的头文件可以避免强制重新编译。这种实现的另一个问题是，typedef 很软弱[Wilson03]，没有提供任何类型安全（type safety）。尽管有这些潜在的缺点，Turbine 对于 typedef 的 d 关键字还没有重大的问题。

游戏对象的Qualities	
键	数值
Health_Max_IntStat	100 (integer)
Health_Current_IntStat	100 (integer)
XP_Value_IntStat	1700 (integer)
AttackDelay_FloatStat	5.5 (float)
UsesMagic_BoolStat	true (boolean)

图 6.3.1 存储在键值对中的样本

```
// IntStat.H

typedef int IntStat;
IntStat Undef_IntStat = 0;

// IntStatList.H
```

```
#include "IntStat.H"

IntStat Health_Current_IntStat = 1;
IntStat Health_Max_IntStat = 2;
// 等等
```

给定了关键字和数据类型两者的定义，下一个要讨论的问题则是数据要存储在哪里。Qualities 由两部分组成：每个对象类型缺省数据和每个对象实例数据（见图 6.3.2）。指定的世界对象的缺省值保存在 Qualities 对象“defaults（缺省）”中。每个对象维护一个“local（本地）”Qualities，充当所有从缺省值变化数据成员的仓库。未改动过的任何数据不存储于本地某个对象中。仅对修改后的数据成员进行存储是 Qualities 模式的关键，这样使得 Qualities 能对对象状态内存的消耗达到最小。

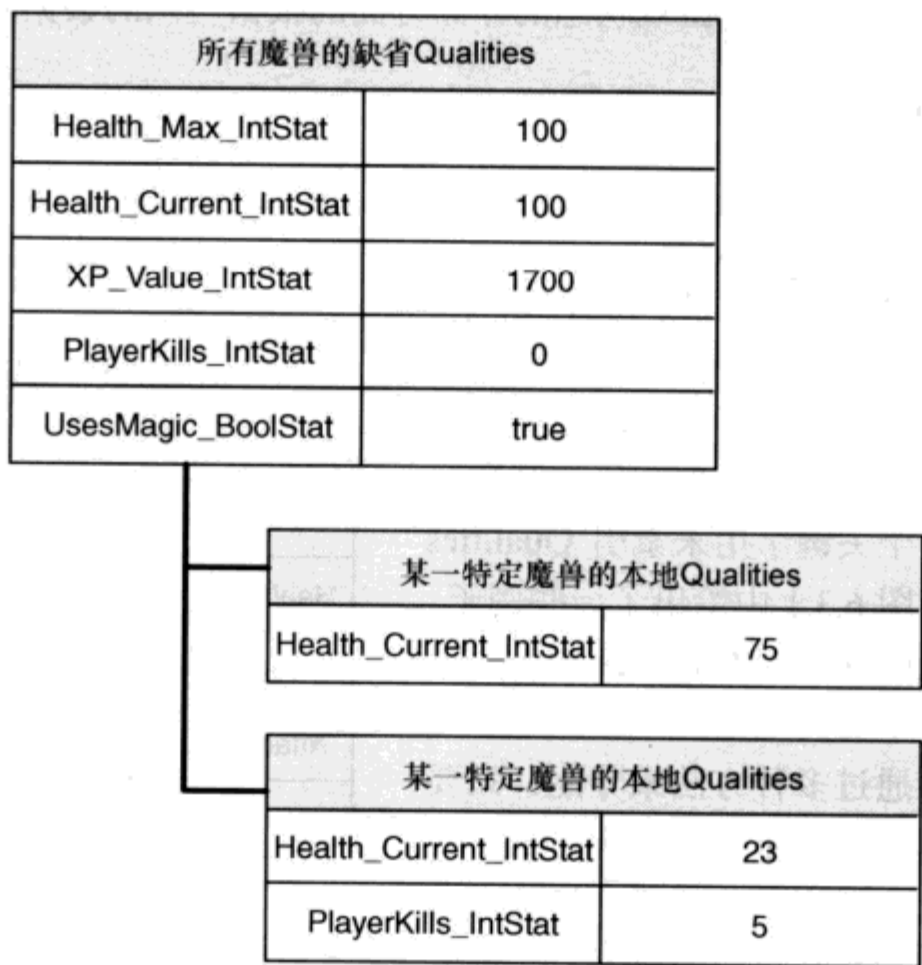


图 6.3.2 关于两个魔兽的 Qualities 结构说明

对于对象的缺省 Qualities 是由内容设计师定义的，并由后端数据存储机制作预处理。在运行时，这些值作为一个 singleton 操作的每个服务器全局对象的要求来装载 [Gamma95]。缺省的 Qualities 对象为游戏对象提供了缺省的数据成员的访问。作为一个 singleton 来操作意味着全局的缺省 Qualities 对象通过延迟一个对象缺省状态的装载直到它被请求时，使得内存消耗最小化。一旦载入，在内存中只需要一个拷贝，因为每个请求将按照特定路线发送到被装载的版本。

6.3.3 Qualities API

如果没有一种方法来进行访问和修改，所有游戏世界中的数据是无价值的。Qualities

API 可以归为两类：值查询（Value query）和值调整（Value adjustment）。Qualities 允许游戏逻辑收集一个状态的当前值。值调整分成更更多的子类别：值设定（Value set）和值重置（Value reset）。值设置是将状态修改成一个特别的值并将这个新的值加入对象的本地 Qualities，而值重置则是将状态的值重新设为该对象的缺省值并将其从本地的 Qualities 中删除。

当游戏逻辑想要知道一个给定的数据值，查询首先检查对象的本地 Qualities。如果一个值在本地存储，那么那个值就会被返回；如果没有，则查询对象的缺省 Qualities。访问 Qualities 的 API 存储在游戏对象层次的最高层。这些 API 为整个游戏代码库的 Qualities 接口提供简单的包裹（wrapper）。API 的配置是引擎结构特定相关的，所以对别的游戏引擎而言，API 放在别的地方会更有意义些。

```
class BaseGameObject
{
public:
    bool GetInt(IntStat intStat, int32& val) const {
        return m_qual.GetInt( intStat, val );
    }
    bool SetInt(IntStat intStat, int32 val) {
        return m_qual.SetInt( intStat, val );
    }

private:
    Qualities m_qual;
};
```

Qualities 的内部结构非常简单。对每种数据类型，一个联系关键字 typedef 到数据类型哈希表按照对于这个对象基数值的 ID 来存储。当对象被创建时，该值就会被初始化，同时要保证魔兽的每个实例都有一个数据连接缺省的魔兽 Qualities。

缺省的 Qualities 和本地 Qualities 一样使用相同的类，它们的主要不同在于数据成员的初始化。本地的 Qualities 是由 gameplay 代码创建的，而缺省的 Qualities 是由游戏的内容库来创建的。

```
typedef hash_table< IntStat, int32 > IntStorage;
// 对每种数据类型重复同样步骤

class Qualities
{
public:

    // 主要的访问器
    bool GetInt(IntStat intStat, int32& val) const;
    bool SetInt(IntStat intStat, int32 val);
    //对每种数据类型重复同样步骤

    // 这是缺省的 Qualities 结构吗？
    inline bool IsDefaultQualities() const {
```

```

        return (m_BaseQual == INVALID_DATABASEID);
    }

private:

    // 我们缺省值的 ID
    DatabaseID    m_BaseQual;

    // 数据类型特定的存储
    IntStorage*   m_IntStore;
    FloatStorage* m_FloatStore;
    BoolStorage*  m_BoolStore;
};

```

当查询某个值时，Qualities 首先检查本地的 Qualities。如果在本地没有查到，那么就查询请求对象的缺省 Qualities。

```

bool
Qualities::GetInt(IntStat intStat, int32& val) const
{
    // 检查我们是否有该数据类型的本地散列表
    // 如果有的话就检查值的本地版本
    if (m_IntStore != NULL &&
        (m_IntStore->Lookup(intStat, val))) {
        return true;
    }

    // 如果是省缺的 qualities，既然所请求的值没有存储
    // 那么就停止
    if (IsDefaultQualities()) {
        // 断言，日志错误并中止
        return false;
    }

    // 现在我们需要检查缺省的 Qualities
    // 数据库的调用应该用代码库 (codebase) 的 singleton 访问方法来替换
    Qualities* baseQualities =
    Database::GetBaseQual(m_BaseQual);
    if (!baseQualities) {
        // 断言，日志错误并中止
        return false;
    }

    // 返回缺省 Qualities 中的值
    return baseQualities->GetInt(intStat, val);
}

```

运行时在 Qualities 中设置一个值比较简单。如果没有本地的存储空间，那么一定要分配一个。然后调用哈希表增加该关键字值对。

```

bool

```

```
Qualities::SetInt(IntStat intStat, int32 val)
{
    // 确定是一个本地 Qualities 的拷贝
    // 而不是缺省的 Qualities
    if (IsDefaultQualities()) {
        // 断言, 日志错误并中止
        return false;
    }

    // 如果需要的话, 创建一个新的本地存储
    if (m_IntStore == NULL) {
        m_IntStore = new IntStorage();
    }

    // 实际上在本地存储值
    return m_IntStore->AddValue(intStat, val);
}
```

6.3.4 使用 Qualities 的好处

使用 Qualities 模式有很多的好处。根据写入的数据进行复制的方法中, 固有的实时内存分配意味着多个一致对象的内存需求明显减少。每个游戏对象只需在本地存储对于对象惟一的数据。当世界状态需要被保存时, 保存对象状态最有效的方法就是只保存不同于缺省值的状态。保存每个游戏对象的本地 Qualities 只自动保存了与缺省对象值不同的数据。Qualities 不需要重复查看对象的状态就能知道它和缺省对象是否不同。

Qualities 在 gameplay 逻辑和 gameplay 状态之间提供了清晰的接口。由于只展现了一个狭窄的接口给 gameplay 系统, 这就使得 gameplay 代码更为清晰。通过把这两者分开, Qualities 的数据结构和内部代码路径能够无需使用那些 API 的代码修改就可以被修改。当 Qualities 的结构被改动时, 这种分开也减少了编译次数。

通过保存和对象相分离的缺省值, Qualities 考虑到了对游戏世界中每个实例化对象的简单修改。通过在全局 Qualities 中更新数据值, 由于值和每个对象实例不是一起存储的, 游戏世界中的每个对象就能自动引用新的值。这意味着要更新游戏中每支长剑的缺省损害, 设计师就只需要在游戏内容库中更新一条简单的条目, 而不是运行一条脚本来检索存储的整个游戏状态并更新每个对象实例。当游戏数据改变时, 这就减少了服务器停机时间, 这一点一直是玩家所想要的。

6.3.5 结论

当游戏继续支持越来越多的玩家时, 新的挑战自己会暴露出来。当规划一个大规模的游戏时, 内存使用和游戏状态存储的大小是要考虑的两个重要因素。这种 Qualities 框架提供了一种根据写入数据进行复制的方法的机制来处理某些与规模相关的问题。在 Turbine 引擎中使用 Qualities, Turbine 已经取得了非常好的效果。

6.3.6 参考文献

[Gamma95] Gamma, Erich, et al., *Design Patterns: Element of Reusable Object-Oriented Software*, Addison-Wesley Publishing Co., 1995.

[SNIA03] Storage Networking Industry Association, "A Dictionary of Storage Networking Terminology: Common Storage Networking-Related Terms and the Definitions Applied to Them," available online at www.snia.org/education/dictionary/c/#copy_on_write, 2003.

[Wilson03] Wilson, Matthew, "True typedefs," March 2003, *C/C++ Users Journal* (March 2003), pp. 35–38.



6.4 在客户/服务器环境下运用并行状态机

作者: Jay Lee, NCsoft Corporation

E-mail: jlee@ncaustin.com

译者: 许竹钧

审校: 万太平

在[Alexander03]一文中, 作者给出并行状态机 (Parallel-State Machine, PSM) 的概念介绍以及它们用于创造更加逼真和更有吸引力的游戏角色。通过并行方式协调几个简单的状态机, PSM 思想允许比任何一个简单状态机本身所能表示的更复杂角色行为的建模。

[Alexander03]中所讨论的 PSM 包含了三个层或状态机来描述角色的行为, 分别是姿势 (Posture)、移动 (Movement) 和动作 (Action)。那个方法被真实反映到这里。Posture 状态机处理一个角色在一个世界中怎样能够看起来更加真实, 包括各种姿势, 比如直立、游泳、盘旋或骑马。Movement 层处理一个角色在游戏世界中的运动。这包括对于每个明显运动方向的状态: 向前、向后、往左和往右, 再加上角色休息时的停止状态。Action 层则处理各种在游戏中一个角色所能做的行为。这些独立于其他两层的状态可能出现。例如, 一个角色可以在游泳的同时, 用武器攻击某个怪物。

PSM 可以为各种游戏子系统所重用, 如人工智能 (AI)、用户界面、咒语效果和动画系统。有意思的是, 每个子系统只需要依靠 PSM 中的状态就能保证相互一致。这样的话, 在这些子系统中就只需携带较少的状态标志位或复杂的条件码。

本文阐述了在客户/服务器环境下 PSM 的使用。其中 PSM 在保持客户端和服务端同步性的同时, 驱动由 AI 和玩家控制的角色的看得见得行为。该代码是用 Python 写的, 但很容易将其中的理念应用到你所熟悉的语言中。

6.4.1 独立状态

PSM 的基石就是状态。在引入[Gamma95]中关于状态模式 (State pattern) 方面的概念之后, 状态就被模拟了。

每个状态共享以下的接口。

```
def GetStateId():  
    # returns a unique value for the state
```

```
def GetStateTypeId():
    # returns the layer the state belongs to

def OnEnterState(actor):
    # setup code executed when entering this state

def OnExitState(actor):
    # teardown code executed when exiting this state
```

下面的代码段说明了状态之间的转换过程。

```
def Transition(self, actor, newState):
    self.currentState.OnExitState(actor)
    self.currentState = newState
    self.currentState.OnEnterState(actor)
```

在每次转换时，每个离开状态（outgoing state）会相应结束，然后被新进入状态（incoming state）所替换，再执行它的 **setup** 代码。例如，假如某个角色改变它的姿势行为为坐下，渲染使它不能移动，然后 **Sitting** 状态应该在它的入口方法那里实现那个特征。

状态模式中有一条指南是每个状态应该使用 **Singleton** 模式[Gamma95]来实现，以使内存使用量最小。任何给定的状态只应该作为一个单独的实例退出，同时拥有指向 singleton 的引用状态的 **Client** 端来表明当前正处于那个状态。Python 为 singletons 的实现给出了一种简便的机制——即模块（**Module**）。模块是在一 Python 文件范围内的一种封装。需要引用一个状态的地方都要有指向给定模块的引用。

以下是一个作为模块来实现的状态的范例。

```
# dead.py

import shared
import shared.characterstatedata as _csd

# register this state for convenient lookup
# from id value to state module
shared.RegisterState(__name__)

def GetStateId():
    return _csd.DEAD

def GetStateTypeId():
    return _csd.POSTURE

def OnEnterState(actor):
    # when dead, block movement
    actor.BlockMovement()

def OnExitState(actor):
    # moving out of dead, unblock movement
    actor.UnblockMovement()
```

当状态是 **singleton** 并且没有任何实例数据的时候，它们通过传递的参数取得上下文环境（**context**）来进行操作。**OnEnterState** 和 **OnExitState** 方法取得对于正在操作的参与者（**actor**）的引用。这就允许一个状态在一个角色的特定实例上进行处理。在这个例子中，当角色在

Posture 层中进入死亡状态时，在 Movement 层上和那个角色相关的任何转换都会被中断。当退出死亡状态时，在 Movement 层上有关那个角色的转换接下来就不再被中断。每个状态需要在进入时实现所期望的游戏规定，在退出时取消那些规定。

当系统进一步地完善，增加新的状态依然简单，也不需要 PSM 的理解做任何的改变。只需要（为状态的加入）选择恰当的层，同时确保新的状态在所选择层上和其他的状态没有重名。要多注意和其他状态匹配不太好的状态，特别是名字听起来像是其他状态所结合的那些状态。

要注意的是，为每个状态赋一个惟一的 ID 是出于长远考虑。这就为 PSM 的各种客户端提供了数据驱动方法的基础，这些稍后会在本文中进行介绍。

6.4.2 角色状态管理器

CharacterStateMgr 是管理员（manager）类，负责将每个状态集合起来作为独特的、协调的状态机或层。

一旦被创建，一个 CharacterStateMgr 将作如下初始化：

```
self.__stateMachine = {}
self.__stateMachine[_csd.POSTURE] = _standing
self.__stateMachine[_csd.MOVEMENT] = _stopped
self.__stateMachine[_csd.ACTION] = _idle
self.__blockedState = {}
```

第一个字典，self.__stateMachine，表示将每一层或独特的状态机对于当前状态层的关键值对。每个状态机初始化为恰当起始状态。

第二个字典，self.__blockedState，管理了那些当前被中断的转换状态。很明显，状态转换的中断是靠引用的计数机制来实现的，其中关键词是对状态的引用，而值就是当前在给定状态上的中断个数。这就使得多个子系统在保持良好总体期望效果的同时，还可以调用 Block() 和 Unblock()。

举例说明一下。不同时期的两个魔法被释放在同一个角色上，两者都阻塞该角色转换到 Posture 层的 Fighting 状态。两者都在时，这个角色就无法战斗。当时效短的魔法作用消失，但由于另一个魔法的阻塞依然起作用，该角色还是不能够战斗。只有当第二个魔法作用消失后，才能实现到战斗状态的转换，而这才是所期望的行为。

CharacterStateMgr 类提供了一些方法来实现以下功能：

- 转换到一个状态，同时注意中断机制；
- 转换到一个状态，但不检查中断情况；
- 访问各个状态层和它们的值；
- 中断和释放独立的状态；
- 中断和释放状态组。



详细内容请参考配套光盘文件中文件 characterstatemgr.py 中的源代码。

6.4.3 使用 CharacterStateMgr

CharacterStateMgr 是一个“混合类型 (mixin)”类，是用来“混合”任何需要通过继承而来的角色状态管理器服务的类。给定一个叫做 Actor 的类，设计用来表示游戏中玩家角色和 NPC (NonPlayer Character) 的基类，它是从 CharacterStateMgr 继承而来的。这样的话，每个 Actor 就是一种 CharacterStateMgr，同时通过每个 Actor 的实例可直接访问 PSM 实例。同样地，客户和服务端两者都是以相同的行为来从 CharacterStateMgr 上继承。

6.4.4 保持客户端和服务端端的同步

注意 CharacterStateMgr 的全部 source (发起者) 和相关联的状态是与平台无关的。也就是说，在 source 中不存在客户端或服务端相关的代码。在使用 CharacterStateMgr API 的同时，是由 PSM 之外的系统来实现平台特定的代码。

考虑到每个 Actor 的 CharacterStateMgr 数据的当前值，客户端和服务端保持同步非常重要。这是通过以下方法来实现的。

客户端使用 CharacterStateMgr 中的 RequestTransition() 方法来请求进入一个状态。在客户端机 (client machine) 上，该代码注意到当前已知状态并作相应响应。一旦客户开始移动，它就发送相同的请求给服务器来进行处理。它也调用 UpdateStateDependentSystems()，该方法会通知每个和 Actor 的 PSM 相关的子系统，使得子系统能够正确地处理状态转换。以下代码说明了如何处理客户端所发出向前移动的请求。

```
def HandleMoveForward():
    if actor.RequestTransition(_moveforward):
        actor.MoveForward()
        SendServerRequest("StateChangeRequest",
            actor.GetId(), _moveforward.GetStateId())
        UpdateStateDependentSystems(actorId)
    else:
        DisplayMessage("Unable to move at this time")
```

在服务器端，该请求被接受并处理。服务器端来决定这个转换是否有效，同时尊重使用转换中断机制任何已知的禁令。如果这个转换是允许的，RequestTransition() 方法就会调用 TransitionTo() 来转换到指定的状态。在 CharacterStateMgr() 之外，该状态的改变也会被广播给除发起者外所有感兴趣的客户。之所以不考虑发出请求的客户端是因为它已经处于期望的状态，同时我们也乐意不放过任何能节省带宽的机会。

如果请求被拒绝，那么只有请求发起人才会被通知。举个例子，如果服务器端认定该 Actor 实际上已经死亡，无法移动，那么它就会发送一个状态修正给发起人。其他感兴趣的客户已经在观察服务器端的状态，因此我们同样避免了不必要的通知也节省了带宽。以下代码片断说明了正在处理状态改变请求的服务器。

```
def StateChangeRequest(clientId, actorId, stateId):
```



```

actor = GetActorInstance(actorId)
state = characterstatemgr.GetState(stateId)
if actor.RequestTransition(state):
    BroadcastToAllButSender("StateChange",
        actorId, stateId)
else:
    # send client correction
    SendClientMessage("StateCorrection",
        clientId, actorId, actor.GetAllStateIds())

```

一接收到状态修正，客户端立即和指定的状态集同步。当由服务器端作了修正时，不再进行检查，每个状态毫无疑问地在指定的 Actor 进行转换。对 UpdateStateDependentSystems() 的调用允许了子系统来正确地处理修正。以下的代码片断对此作了说明。

```

def StateCorrection(self, actorId, stateList):
    actor = GetActorInstance(actorId)
    for stateId in stateList:
        state = GetStateModuleById(stateId)
        actor.TransitionTo(state)
        UpdateStateDependentSystems(actorId)

```

当在服务器端检查且转换有效后，除了转换的发起人之外，每个客户接受和处理状态的改变。一旦 Actor 被更新，对 UpdateStateDependentSystems() 的调用则允许其他子系统作正确响应。以下代码说明了在感兴趣的客户端上的 StateChange 句柄。

```

def StateChange(actorId, stateId):
    actor = _GetActorInstance(actorId)
    state = _characterstatemgr.GetState(stateId)
    actor.TransitionTo(state)
    UpdateStateDependentSystems(actorId)

```

由于服务器端的初始化动作，或是在服务器端控制下其他客户端的活动而造成的影响，状态也会在 Actor 上发生改变。例如，NPC 受到服务器端的控制，在感兴趣的客户端看来应该是正确的。下一节的代码给出了一个实例，一个决定向另一个不在攻击范围内的 Actor 前进的生物。状态的改变被广播到所有感兴趣的客户端，并由前面所给的 StateChange() 方法进行处理。由于服务器端独立控制生物的状态，也就不需要检查该转换是否在客户端上有效。任何时候，当状态从服务器端转移到客户端，我们则会考虑服务器端作为授权人，并执行转换。在进行这项工作时，我们相信干扰只发生在未保持同步的客户端身上。只要每个其他的客户端在正确的状态观察到一个受干扰的客户端，那么至于攻击者的感受如何，我们不用理会。

```

def Attack(self, actorId):
    actor = _GetActorInstance(actorId)
    if InRange(self.GetPos(), actor.GetPos()):
        # attack with weapon
    else:
        if self.RequestTransition(_moveforward):
            BroadcastToAll("StateChange",

```

```

        self.GetId(),
        _moveforward.GetStateId())

```

6.4.5 状态依赖的子系统

当一个 Actor 中的状态发生改变，为了正确响应，每个状态依赖的子系统都会被通知。在动画系统中，如这里所示，该响应可能是触发 UpdateAnimation() 方法。

```

def UpdateAnimation(actorId):
    actor = _GetActorInstance(actorId)
    if actor.GetMovementState() == _stopped:
        baseAnimName = GetIdleAnim(actor)
    else:
        baseAnimName = GetMovementAnim(actor)

    actionAnim = None
    if actor.GetActionState() != _idle:
        actionAnim = GetActionAnim(actor)

    PlayAnimations(baseAnimName, actionAnimName)

```

基于当前移动和姿势的状态，动画系统会一直播放基本的动画。要是角色正在休息，那么它会返回一个相应的休息画面。如果角色在移动，就会返回合适的移动动画。这个系统同样允许在一个动画中，身体的不同部位有优先级的区分。如果一个动作画面和一个空闲画面一起播放，那么该动作将由整个角色表现出来。举个例子，一个挥舞武器的角色，所看到的是，它处于挥舞的姿态，并摆出适当的站姿。然而，要是该角色移动的话，那么播放的是身体的上半部分，同时身体下肢会反映出移动。这就能让角色在用武器进行攻击的同时也能移动。

接下来说明的是，对于检索基于 PSM 中当前状态值的各种类型的动画，动画系统如何实现那些方法。

```

idleAnims = {_standing : 'fidget.ani',
             _swimming : 'tread_water.ani'}

moveAnims = {
    (_standing, _moveforward) : 'runforward.ani',
    (_hover, _moveright)      : 'hover_right.ani'}

actionAnims = {_crafting : 'crafting.ani',
               _fighting  : 'fighting.ani'}

def GetIdleAnim(actor):
    # return idle based on posture state
    stateId = actor.GetPostureState().GetStateId()
    return idleAnims[stateId]

def GetMovementAnim(actor):
    # return movement anim based on posture and

```

```
# movement states
posStateId = actor.GetPostureState().GetStateId()
movStateId = actor.GetMovementState().GetStateId()
return moveAnims[posStateId, movStateId]

def GetActionAnim(actor):
    # return anim based on current action state
    stateId = actor.GetActionState().GetStateId()
    return actionAnims[stateId]
```

代码说明了一个重要概念。改变基于 `CharacterStateMgr` 中状态的行为本质可以完全是数据驱动的。因为每个状态有相关联的惟一标示，代码可以通过简单的数据结构来查询结果——在我们的案例中（数据结构）指的是，将一个或多个状态标示映射到一个动画文件的字典。要注意的是，虽然在这里值是手工生成的，但这些值可以放在一个外部的文本文件中，或更好些，由诸如关系数据库之类的集中仓库自动生成。对该概念的进一步探讨可参考[Lee03]。

不难想象其他对 PSM 数据会产生影响的其他子系统，举个例子，如用来结束一个角色当前状态的声音的集合。不考虑 PSM 的应用的话，它为客户端和服务端提供了一致的结果。

6.4.6 结论

本文说明了并行状态机是个简单但功能强大的概念，它明显减轻了复杂的角色状态管理和在客户/服务器环境下的同步所造成的负担。

用于实现状态和 `CharacterStateMgr` 的基本源在客户端和服务端是相同的，这样掌握起来比较简单，也便于维护。在客户端和服务端上的游戏系统能够平衡 `CharacterStateMgr` 来实现它们期望的功能，如果做得有条不紊的话，在共享的服务器端和多个客户端之间则会保持同步。当需要增多，新的特征被添加时，对该结构的补充一目了然，不会引起指数级的增长或管理上的问题。

6.4.7 参考文献

[Alexander03] Alexander, Thor, "Parallel State Machines for Believable Characters," *Massively Multiplayer Game Development*, Charles River Media, 2003.

[Gamma95] Gamma, et al., *Design Patterns*, Addison-Wesley Longman, Inc., 1995.

[Lee03] Lee, Jay, "Leveraging Relational Database Management Systems to Data-Drive MMP Gameplay," *Massively Multiplayer Game Development*, Charles River Media, 2003.

6.5 位打包：一种网络压缩技术

作者：Pete Isensee, Microsoft Corporation

E-mail: pkisensee@msn.com

译者：许竹钧

审校：肖罡

网络游戏通常发出多种数据结构，里面的数据包含了位置、速度、加速度、状态标示，还有其他重要的游戏状态信息。通常，被传送信息的很多位（Bit）并不包括重要数据。例如，一个游戏可能用 32 位的整型数来发送范围在 0~100 000 的位置。每个位置占用了 17 位，但剩下的 15 位就白白浪费掉了。那些没用到的位其实可以很快累积起来，用于大型的数据结构。

网络游戏开发者的烦恼之一就是要在网络中限制数据的流量。本文介绍了一种位打包技术，它把网络数据结构的各个元素压缩到二进制流中，其中流的每一位都包含相关数据。这种技术能够大大减少网络包的大小。该技术的第二个优点在于，能够在调试版本中对网络数据结构的中的元素做范围检查，用来检验它们是否在开发者规定的误差允许范围内。类似的打包技术在现有的网络游戏中已经被使用过了。

6.5.1 一个实例

发明该技术的目的是为了更方便融入到在标准套接字上所发送的现有 C 结构（Struct）和 C++ 类。这里是一个游戏网络代码的简化范例。

```
struct NetData
{
    unsigned char  MsgType;      // 0 - 28
    unsigned long  Time;         // 0 - 0xFFFFFFFF
    unsigned short Element;      // 148~1153
    int            XPosition;     // -100,000~100,000
    int            Yposition;     // -100,000~100,000
};

NetData nd;
send( s, (const char*)( &nd ), sizeof(nd), 0 );
```

在 32 位平台，正在发送和接受的数据量至少为 17 字节。如果该结构不是字节对齐的，那么数据量有可能在 20 字节以上。如果发送者和接收者在不同平台上，那么可能存在字节顺序问题，这需要发送者在发送每项数

据前将其转换成网络顺序，接受者则将每项转换回主机顺序。

这是用位打包技术定义的结构：

```
struct NetData : public BitPackCodec< 5 >
{
    PkUInt<unsigned char,5>          MsgType;
    PkUInt<unsigned long>             Time;
    PkUIntRange<unsigned short,148,1153> Element;
    PkInt<int,-100000,100000>         XPos;
    PkInt<int,-100000,100000>         YPos;
};
```

“可打包的”数据项替换了原先的数据项。这些数据项的用户完全可以将它们和它们的原始整型数一样对待。像正常的整型数一模一样，可以对它们赋值，读和写。可打包项的好处在于它还知道如何将自己打包到比特流中。此外，整型数值的范围已为编译器所知。由于范围由模版参数组成，不存在额外的内存消耗，但进行范围检查可以让编译器发挥更好的作用。

NetData 结构从一个称为 BitPackCodec 的混入类继承而来，它支持了两种关键函数，Pack() 和 Unpack()，还有一个注册函数。注册函数会通知编解码器 (codec) 哪一项会被插入到流中及其出现的顺序。

现在，不再发送原始的 NetData 结构，而是先将它包装到一个位流中。在这个比较特殊的例子中，该位流缓冲区的大小只有 83 个字节，和原来结构的 136 个字节相比，节省了 40%。

```
NetData nd;
int size = nd.GetPackedBytes();
char* pBitStream = new char [ size ];
send( s, nd.Pack( pBitStream ), size, 0 );
```

接收端一旦接收到位流，就将它解压缩成原来的结构格式。

```
nd.Unpack( pBitStream );
```

打包函数自动处理包括字节交换等所有琐碎细节。新的 NetData 结构可以由整型最有效的任何方式组成，而不是非要求助于位域 (Bit field) 或其他在内存中压缩的方法。

6.5.2 难点

关于这种特殊的技术有三个棘手的问题。最难的部分就是位打包本身。虽然写一个每次将一位打包的简单函数只需要几行代码，但将位一个个打包效率比较低。写一个将字节打包的算法要复杂很多。位流、追踪打包内容所要的列表等需要一个独立的缓冲区，所以第二个难点就是如何最小化该技术的内存轨迹。第三个麻烦的问题是如何把这种技术做的足够灵活，使它能方便地融入现有的游戏代码。下一节将对每个问题作详细讨论。

6.5.3 位打包

本文中使用的算法基于对源整型数和目标位缓冲区的排列，因此目标位流的下一个有效

位和源数据的第一个有意义的位是对齐的。每次将一段源比特位拷贝到目标比特位，其中每段是由下一个源或离目标最近的字节边界来定义的。图 6.5.1 给出了一个实例。在目标缓冲区中第一个有效位是位 6。有 13 位需要拷贝。排列以下需要被拷贝的位（阴影部分），很清楚有 4 个拷贝操作，定义为 A、B、C、D。例如，段 B 由位缓冲在第 8 位的第一个字节边界来定义，下一个字节边界在源整型数的第 8 位。

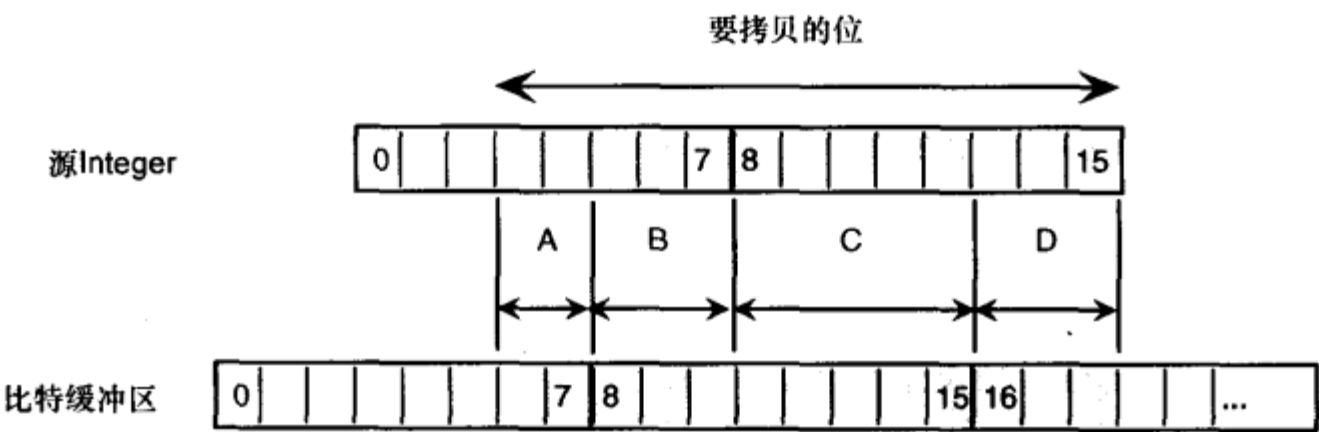


图 6.5.1 位打包算法

如果源整型数和位缓冲的字节边界正好对齐，一次最多可以拷贝 8 位。如果不对齐的话，那么每个字节分两块总共 8 字节进行拷贝。解包的方式完全一样，不过源和目标要对调一下。在将一个整型数的各个位拷贝到位缓冲之前，该整型数被转换成网络格式。在解包阶段，恢复的整型数则转换回主机格式。

打包算法用一个 C++ 的模版函数来实现，它可以打包从 8 位到 64 位任何无符号的整型。函数的原型也一样。为了保持标准套接字函数的兼容性，位缓冲是一个常规的字符指针，而不是无符号的字符指针，大小是个 int 型，而不是无符号的。

```
template< typename T > // T 是任一整型
void Pack( const T& tSrc, int nSrcBits,
           char* pDestBitBuffer, int nDestStartBit );
```

字节交换也是用模版函数来实现的，这就避免了勉强用 switch 语句来决定是用 htonl() 还是 htons()。接下来会对字节交换功能作一下说明。它利用了 STL 的倒置函数。有效的编译器能够优化掉对 IsPlatformLittleEndian() 的调用，并完全将倒置算法内联，使得这些功能高效。

```
inline bool IsPlatformLittleEndian()
{
    const int n = 1;
    return( *((char*)( &n ) ) ? true : false );
}

template< typename T >
T ReverseBytes( T n )
{
    unsigned char* p = (unsigned char*)( &n );
    std::reverse( p, p + sizeof( T ) );
    return n;
}
```



```
template< typename T >
T HostToNetworkOrder( T n )
{
    if( IsPlatformLittleEndian() )
        return ReverseBytes( n );
    return n;
};
```

6.5.4 用于可打包数据类型的通用接口

整型并不是能被有效打包的惟一数据类型。例如，从尾数剥离位的话，还可以对 IEEE 的浮点值进行打包。用多种压缩方式还能打包 String 类型。这里讲的方法主要针对多种情况的整型数，但可以用来对任何数据进行位打包。对打包概念进行抽象的最佳方式是从一个抽象的基类开始。

```
class Packable
{
public:
    virtual ~Packable() = 0;
    virtual int GetBits() const;
    virtual void Pack( char*, int* StartBit ) const;
    virtual void Unpack( const char*, int* StartBit );
};
```

Packable 类包括了其他的有用函数，但这里给出的三个方法是里面的关键方法。GetBits() 返回了 Packable 对象中有意义的位的个数。这些是会被对象来打包和解包的位。Pack() 将 Packable 对象中 GetBits() 方法所得到的位拷贝到输入缓冲区，从指定的位开始。它从 GetBits() 所得到的起始位开始递增。Unpack() 从指定位开始的位缓冲生成一个新的 Packable 对象。它也从 GetBits() 所得到的起始位递增。Packable 接口是为了和套接字接口相兼容而设计的，它用了无格式的旧的字符指针。

6.5.5 用于可打包数据类型的接口

本文对整型进行打包。代码是基于模版的，而不是为每个整型（如 unsigned char、short、long 等）创建一个特定的类。这样，通过三个不同的接口，即支持了所有的整型。

```
template< typename T, int Bits >
class PkUInt : public Packable

template< typename T, uint64 Min, uint64 Max >
class PkUIntRange : public Packable

template< typename T, int64 Min, int64 Max >
class PkInt : public Packable
```

PkUInt 类是为一定数量位的无符号整型所设计的。Caller（调用程序）指定位的数目作

为模版参数。例如，一个网络消息的类型可能用 5 个 bit（值从 0~31，或者是有 5 个重要 bit 的位掩码）存储。PkUinRange 类是为特定值域的无符号整型数而设计的。该范围可以基于零，或者可有一个正的最小值。在任一种情况下，PkUinRange 类都会智能地在数字被打包之前将它格式化，在解包时做反向格式化。例如，如果范围在 1000~1127 内，那么只有 7 位会被用来存储在位缓冲区。PkInt 类是为了特定值域的有符号整型数而设计的。这个类也在数字被打包之前将它格式化并转换成无符号的值。这三个混合类中的每个类都从 Packable 类继承而来，并实现相应的 Packable 函数。构造函数和转换操作符确保了可打包整型数的使用者可以像正常的整型数一样对待它们。由于这些类使用了模版参数，不需要额外的存储来保留位或范围的值。这些类唯一的代价就是 sizeof(T) 加上 Packable 类需要的虚拟函数表的指针。

6.5.6 编解码器

现在我们所需要的就是能方便包装结构、类或者 Packable 对象列表的方式。设计目标则需要方法是透明的并尽可能少用内存。开始谈 BitPackCodec 了。BitPackCodec 所需要的是，打包项的数量、打包的顺序和打包的位缓冲区。虽然位缓冲区能由类做内部维护，但就性能和效率而言代价昂贵。在游戏中，更好的办法是预分配一个足够大的位缓冲区，来处理网络包和根据需要将缓冲区传给编解码器。

当要维护打包项列表的时候，像 std::vector 或 std::list 等灵活的列表数据类型有内存分配的代价，这是我们想要避免的。文章中的解决方案就是将 Packable 对象列表存入 C 风格的数组中。为了得到尽可能大的灵活性和高性能，数组的大小是一个模版参数。这个办法的好处在于它用了最少的内存，不需要堆分配，而且非常快。在调试 build 时，类可以检查以确保数组没有越界。

```
template< int N >
class BitPackCodec
{
private:
    Packable* mPackList[ N ];

public:
    void Register( Packable& );
    char* Pack( char* ) const;
    void Unpack( const char* ) const;
    int GetPackedBytes() const;
};
```

BitPackCodec 是作为混合类来设计的。对于任一个包含多个 Packable 对象的结构 X，X 从 BitPackCodec 继承而来。Register() 将项加入到可打包项列表中。Register() 目的是由 X 的构造函数来调用。这里是一个是用 NetData 结构的例子。

```
NetData() // NetData ctor
{
    Register( MsgType );
    Register( Time );
    Register( Element );
}
```

```
    Register( XPos );  
    Register( YPos );  
}
```

Pack() 和 Unpack()做了所有的工作。它们需要一个大小至少为 GetPackedBytes(), 由调用者分配的外部缓冲区。这里是 Pack()函数的实现:

```
char* Pack( char* pPackBuffer )  
{  
    memset( pPackBuffer, 0, GetPackedBytes() );  
    int nCurrBit = 0;  
    for( int i = 0; i < N; ++i )  
        mPackList[i]->Pack( pPackBuffer, &nCurrBit );  
    return pPackBuffer;  
}
```

Pack()将位缓冲区清空, 然后调用 Packable::Pack()将每个注册的 Packable 项打包到位缓冲区内。编解码器对象可以使用任何实现了 Packable 接口的对象。这种设计的灵活性允许了新的数据类型能根据需要, 让自己可以被打包。

6.5.7 评价折衷

至今为止, 对位打包的最大好处在于减少了网络带宽。当然还有其他的好处。网络数据结构不需要特别的队列、位域, 或者是少见的大小固定的数据类型。在调试模式中还有额外的对打包值的范围检查。最后, 对于不经意的偷窥者, 打包也有助于将信息模糊。

对位打包的代价则有三方面: 可读性、性能和内存。发送一个网络结构很简单, 也容易理解; 但将结构打包到位流里增加了复杂度, 并使得调试网络包更加困难。就性能而言, 对位打包和解包是 $O(N)$ 的操作, 其中 N 是被打包项的个数。就 CPU 的代价来说, 考虑到大多数当代游戏所送的相当小的网络包, 还是相当低的。前面所示的类的内存影响几乎可以忽略不计。可打包的整型是 `sizeof(T)` 加上指向 Packable 虚拟函数指针表的指针。Codec 类引入了大小为 N 的一个数组, 同时游戏必须支持和最大网络包的大小一样的位缓冲区。总而言之, 由于明显减少了带宽流量, 这些不失为不错的折衷。

6.5.8 改进

本文包括了对 8 到 64 位整型数打包的代码。这个基本的技术可以延伸, 为所有其他的游戏数据服务, 包括浮点数和字符串。位打包是一种适合网络数据、比较不错的无损压缩技术。但是, 你可以做得更好。如果你对更好的方法有兴趣, 参考一下[Blow03]和[Nelson96]中的算术编码器, 当然还有其他的压缩技术[Bloom]。

6.5.9 结论

位打包完成了以下目标。

- 将数据结构打包到仅包含“重要”位的流中。丢弃所有没用的位。最后节省的带宽相当可观。
- 很容易融入到现有的代码。可以把现有的网络数据结构当作包含原始数据一样。
- 允许网络数据结构对齐来加快速度而不是为了大小。
- 尽可能地透明。
- 对性能的影响最低。
- 内存开销最低。

6.5.10 参考文献

[Bloom] Bloom, Charles, “Compression Algorithms,” available online at www.cbloom.com/algs/index.html.

[Blow02] Blow, Jonathan, “Packing Integers,” *Game Developer Magazine* (May 2002): pp. 16–19.

[Blow03] Blow, Jonathan, “Using an Arithmetic Coder: Part 1,” *Game Developer Magazine* (August 2003): pp. 14–18.

[Frohnmayr00] Frohnmayr, Mark, and Tim Gift, “The TRIBES Engine Networking Model,” available online at www.gdconf.com/archives/2000/frohnmayr.doc.

[Nelson96] Nelson, Mark, and Jean-Loup Gilly, *The Data Compression Book, Second Edition*, M & T Books, 1996.



6.6 多服务器网络游戏的时间和同步管理

作者: 石卫东 (Larry Shi) 和 Tao Zhang, Georgia Institute of Technology

E-mail: shiw@cc.gatech.edu, zhangtao@cc.gatech.edu

译者: 石卫东 (Larry Shi)

多人在线角色扮演游戏 (MMORRG) 自从 *Ultima Online* 的成功以来变得越来越流行。几乎所有的商业 MMORRG 都使用客户服务器构架。在多数情况下采用的是简单的单服务器多用户的方式。为了支持大量玩家同时在线, 开发者通常把一个网上世界分割成多个独立的区域。这样每个区域都小到足够用一个服务器来处理。在区域之间, 很少或没有同步或互动。这样的做法会遇到许多问题, 如果开发者不能把一个网上世界划分成许多很小的区域来保证同一个区域里不会有太多的玩家, 这样的做法就会遇到许多问题。为了保证游戏的可扩展性, 有时不得不用多个服务器来处理一个虚拟区域。但使用多个服务器会带来许多新问题, 一个是时间管理, 还有一个是同步管理。在这篇文章里, 我们将尽可能详细地讨论如何处理这两个问题。

6.6.1 为什么需要时间和同步管理

在一个以多服务器为基础的角色扮演游戏中, 服务器通常用一个后台的数据分流网络连接。每一个服务器用事件驱动的方式来模拟一组游戏实体。实体可以是建筑物、魔兽、玩家、或者宝物等等。如果游戏设计人员能够保证在服务器之间没有互动, 时间和同步管理就不是必须的。这是因为每一个服务器都可以独立地更新所模拟的实体状态而不必查询其他的服务器。如果服务器之间存在互动, 而不用同步管理, 奇怪的事情, 比如已经死去的魔兽也可以开枪等就会在游戏中出现。这样的问题通常是由网络传输延迟造成的。其结果就是, 不同的服务器之间没有一个同步的游戏状态。如果所有的游戏事件都有一个时间记号, 而且每个服务器都根据一个统一的事件系列 (基本事件的时间记号) 处理所有的游戏事件, 那么服务器之间就不难保持同步。由此看来, 时间管理和同步是分不开的。

6.6.2 时钟同步

管理时间第一步要做的是如何让服务器之间, 服务器和客户端之间有一个同步的时钟。有很多做法可以用来同步时钟, 比如网络时间协议。时

钟同步不是我们讨论的重点。我们假设服务器和客户以及服务器之间已经采用了同步的时间。

6.6.3 同步和响应

即时响应是许多游戏设计者想要的，但是同步（consistency）和响应（responsiveness）是一对矛盾的特性，通常不可能兼得。如果服务器和客户端根据玩家的输入或指令立即更新游戏状态，这样做可以得到最即时的响应。但是通常会造成服务器和客户端，服务器和服务器之间游戏状态的不同步。如果要保证最佳的同步，服务器就不能马上处理收到的玩家事件。要保证同步，服务器只能在玩家事件被所有的服务器都收到，而且确定所有的服务器处理都会以同样的序列来处理玩家事件后，才可以对玩家事件进行响应。这会造成很慢的响应。

6.6.4 用多时间管理来一石二鸟地实现同步和响应

对同步和响应的不同要求可以用不同的时间和同步管理方式来实现。已知的时间和同步管理方式可以被分为“保守的”和“乐观的”两种。保守的方式可以用来实现绝对的同步，但是不保证响应，因此不太适合网络游戏这样的实时系统。保守的方式通常被用在小规模的对点方式的游戏中。传统的乐观的方式使用回滚（rollback）的方式来实现模拟状态的同步，但是这种方式会造成一时的可察觉的不同步状态，这使得这种技术同样不适用于游戏。一种相关的同步技术是死亡回忆（dead-reckoning）。这种技术使用基于历史的预测来实行模糊同步。死亡回忆可以实现很好的实时响应，但不能保证很强的同步。在这里，我们根据实体特性的需要采用不同的同步管理方式。比如实体的位置特征，通常并不需要很强的同步性；而实体的经验值、死活等，就需要较强的同步性。对于那些对同步性要求不高度特性，可以采用类似死亡回忆预测的方式来更新。客户端在收到玩家的输入/指令后，可以即时地更新实体状态。服务器在收到用户的输入后，会计算一个正式的更新状态，并发给客户端。如果这两个状态存在差异，客户端可以采用正式的状态来替换不正确的状态，或者以平滑的方式渐进地逼近正式的状态。对于那些要求同步较高的特性，我们可以采用类似保守的方式来更新。我们把这种方式叫做多种时间的管理，因为它既使用保守的时间和同步管理，又使用预测技术。预测技术在这里保证即时响应，而保守方式提供所需的同步性。

6.6.5 实现

实现多时管理，我们作以下假设。

- 虚拟世界被划分为多个区域，每个区域使用一组服务器。
- 服务器连接使用后端局域网并且支持多播（multicast）。
- 对于每个玩家事件，客户端都产生一个时间记号。时间记号也可以在服务器端产生，但这样会对玩家造成不公。
- 客户端和服务器，服务器和服务器之间使用可信赖的传输途径。我们可以用可信赖的UDP（reliable UDP）。

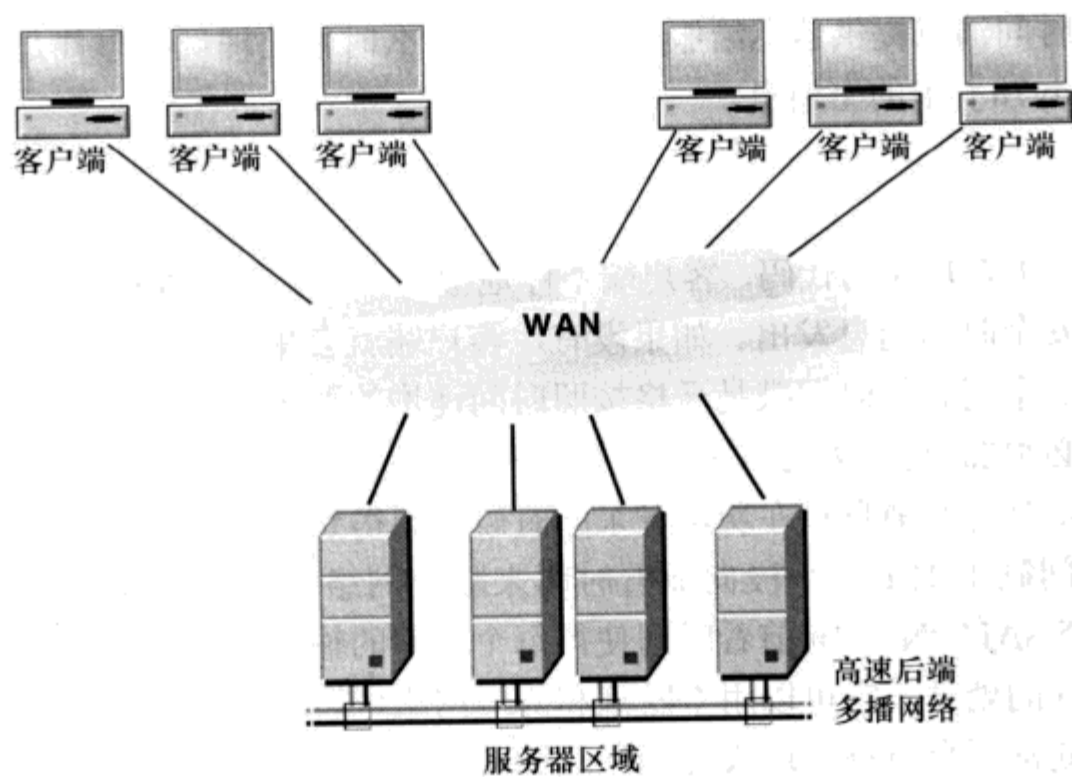


图 6.6.1 多服务器体系结构

下面我们给出使用多线程的伪代码。多线程并不是必须的但可以更好地理解所用的算法。每一个客户或者服务器都使用两个线程。一个是主线程，另一个是时间管理和通信线程。在客户端，主线程处理玩家输入、实体模拟、状态预测、渲染等等。在服务器端主线程实现游戏基本逻辑和实体模拟。

客户端和服务端之间消息的数据结构定义如下。

```
typedef MSG_t struct
{
    DWORD type;           // 消息类型
    DWORD client_id;      // 客户端标识符
    DWORD server_id;      // 服务器标识符
    struct time_t timestamp; // 消息时间戳
    struct time_t safe_time; // 客户端/服务器安全时间
    struct user_input_t user_input; // 用户输入指令
    ...
} MSG_t;
```

表 6.6.1 列出不同的消息类型和它们的目的。

表 6.6.1 消息类型和解释	
类 型	目 的
SERVER_SAFE_TIME	服务器安全时间
CLIENT_SAFE_TIME	客户端安全时间
CLIENT_INPUT	客户端带时间戳（time-stamped）的输入/指令
PREDICTIVE_STATE_UPDATE	预测的新状态
CONSERVATIVE_STATE_UPDATE	保守计算得到的新状态

客户/服务器的安全时间消息是用来驱动保守时间管理的。在保守时间管理下，每个事件都要严格按照时间戳顺序处理，并且保证在处理某个事件以后不会再收到有一个更小的时间

戳的事件。安全时间消息提供信息给服务器来决定什么时候处理玩家事件是安全的。在功能上，类似 [Chandy78] 中的空消息。

1. 客户端

这里，我们给出客户端伪代码，客户端不停地检验在过去的 MAX_NOTIFY_INTERNAL 里，是否有一个安全时间消息发出。如果没有，客户端就会生成这样一个消息并发送给它所连接的服务器，这个消息的时间戳是严格按照时间递增的要求产生的。在通常状态下，安全时间消息并不是必须的，这是因为每个客户端所发出的玩家事件都相当于一个安全时间消息。不带玩家事件的安全时间消息只在玩家发呆的时候才用得到。

客户端在时间戳上加上一小段向前看时间来隐藏网络传输所造成的延迟（伪代码中的 DELAY_COMPENSATION）。向前看时间使得每个玩家的输入都好像发生在将来。这样客户端可以推迟对事件的处理。它可以用来隐藏和降低网络延迟的影响。众所周知，100 毫秒左右的向前看时间通常不会对玩家造成可察觉的影响，但是网络延迟通常都在 100~600 毫秒之间。这样，单单用向前看时间并不能解决网络延迟和同步的问题。表 6.6.2 描述了客户端所使用的时间参数。

表 6.6.2 伪代码用到的常数意义

名 称	意 义
MAX_NOTIFY_INTERVAL	同一客户发出的消息间的最大间隔
DELAY_COMPENSATION	对客户输入/指令所作的延迟时间
MAX_SAFE_TIME_UPDATE_INTERVAL	服务器计算全局安全时间的最大间隔
SERVER_LOOKAHEAD_TIME	加到保守计算下产生的新事件的处理延迟时间

客户端的时间管理/通信线程。

```
while (1)
{
    //用时间标识的消息输出队列，输出消息
    MSG_t* msg;
    msg=out_msg_FIFO->pop_front();
    while (msg)
    {
        msg->timestamp = now + DELAY_COMPENSATION;
        msg->safe_time = msg->timestamp;
        send_msg(msg);
        last_safe_time = msg->safe_time;
        //以消息时间戳排序的任务队列
        work_queue->enqueue(msg);
        msg = out_msg_FIFO->pop_front();
    }
    if (now + DELAY_COMPENSATION -last_safe_time >
        MAX_NOTIFY_INTERVAL)
    {
        msg          = new_msg();
        msg->type      = CLIENT_SAFE_TIME;
        msg->client_id = self_id;
```



```

        msg->safe_time = now + DELAY_COMPENSATION;
        send_msg(msg);
        last_safe_time = msg->safe_time;
    }
    //接收服务器状态更新消息
    MSG_t* msg;
    while(msg = nonblocking_receive_from_server())
    {
        work_queue->enqueue(msg);
    }
    // ...
}

```

客户端的主线程。

```

while (1)
{
    UserInput* new_input = collect_user_input();
    MSG_t* input_msg      = new_message();
    input_msg->type        = CLIENT_INPUT;
    input_msg->user_input = *new_input;
    //将消息压入输出队列
    out_msg_FIFO->push_back(input_msg);

    //返回其时间戳小于当前时间的最早消息
    //如果不存在, 返回空
    MSG * cur_msg = work_queue->dequeue(now);
    //处理用户输入和服务器更新
    while(cur_msg)
    {
        if(cur_msg->type == CLIENT_INPUT)
            client_predictive_simulation(cur_msg);
        else
            process_server_updates(cur_msg);
        cur_msg = work_queue->dequeue(now);
    }
    // ...
}

```

2. 服务器端

现在让我们来看服务器端。每个服务器端有两个时间戳事件队列：一个预测队列用来处理对同步要求不高的消息；另一个是安全队列，用来处理需要严格同步的消息。预测队列使用当前的服务器时钟。对于每个事件，如果它的时间戳小于当前的服务器时钟，服务器就会马上处理它。这不同于安全队列。安全队列的事件处理时间是基于服务器收到的全局的最小安全时间。这个时间是服务器从其他服务器和客户端收到的最小时间，包括当前服务器的安全时间。在每个模拟计算周期，在安全队列中，所有事件如果其时间戳小于全局最小安全时间，都可以被安全地处理而不必担心造成不同步。需要提醒的是，在安全模拟中，模拟时钟是安全时间而不是服务器的真实时钟。

另一个棘手的问题是，安全模拟可能会产生新的带时间戳的事件。比如当击中一个怪兽时可能触发一个新的事件。新事件的时间戳应该是当前的安全时间再加上一个小的时间常量 (SERVER_LOOKAHEAD_TIME)。这个小的时间常量的用处是避免安全模拟中可能出现的死锁。进一步，采用这个时间常量可以提高安全模拟的效率。读者可以参考文献 [Fujimoto98]。

服务器的时间管理/通信线程。

```
while (1)
{
    MSG_t* msg;
    if(now - last_safe_time_update >
        MAX_SAFE_TIME_UPDATE_INTERVAL)
    {
        // conservative_queue->get_minimum()
        //如果队列非空,
        //返回队列中最小时间戳
        //否则返回最大时间戳
        local_safe_time = MIN(client_safe_time_array->
            get_minimum(), now, conservative_queue->
            get_minimum()) + SERVER_LOOKAHEAD_TIME;
        server_local_safe_time_array[self_id] =
            local_safe_time;
        global_safe_time =
            server_local_safe_time_array->get_minimum();
        last_safe_time_update = now;
        if(local_safe_time != old_local_safe_time)
        {
            msg = new_msg();
            msg->type = SERVER_SAFE_TIME;
            msg->server_id = self_id;
            msg->safe_time = local_safe_time;
            reliable_multicast_to_other_servers(msg);
            old_local_safe_time = local_safe_time;
        }
    }
    //处理所有可能的消息
    while(msg = nonblocking_receive_msg())
    {
        if(msg->type == SERVER_SAFE_TIME)
        {
            server_local_safe_time_array[msg->server_id] =
                msg->safe_time;
        }
        else if (msg->type == CLIENT_SAFE_TIME)
        {
            client_safe_time_array [msg->client_id] =
                msg->safe_time;
        }
        else if (msg->type == CLIENT_INPUT)
        {

```

```

        client_safe_time_array [msg->client_id] =
            msg->safe_time;
        //以消息时间戳排序的队列
        predictive_queue->enqueue(msg);
        conservative_queue->enqueue(msg);
    }
    else if (msg->type == PREDICTIVE_STATE_UPDATE)
    {
        predictive_queue->enqueue(msg);
    }
    else if (msg->type == CONSERVATIVE_STATE_UPDATE)
    {
        conservative_queue->enqueue(msg);
    }
}

```

服务器的主线程。

```

while (1)
{
    MSG_t* cur_msg;
    cur_msg = predictive_queue->dequeue(now);
    while(cur_msg)
    {
        server_side_predictive_simulation(cur_msg);
        cur_msg = predictive_queue->dequeue(now);
    }
    //类似 dequeue 但是并不删除消息
    cur_msg = conservative_queue->
        peek(global_safe_time);
    while(cur_msg)
    {
        server_side_conservative_simulation(cur_msg);
        conservative_queue->dequeue(global_safe_time);
        cur_msg = conservative_queue->
            peek(global_safe_time);
    }
    // ...
}

```

另一个需要注意的是，服务器只能在处理完一个事件后，才可以把它从安全队列中清除。这是为了避免进程状态切换可能造成对同步的影响。这里我们给出服务器端伪代码。注意伪代码并没有显示可以用预测计算的消息和安全模拟消息的区别。预测模拟和安全模拟都处理玩家事件并更新那些不需要很高同步要求的实体特性。但安全模拟可以更新所有的实体特性，包括那些有较高同步要求的特性。另一个区别是，只有安全模拟才可以产生新的事件。如果预测模拟也可以产生新的游戏事件，这可能会造成某些奇怪的现象。简单的办法是不要让预测模拟产生新的事件。其他需要注意的问题是客户端死机或者是长时间的网路重传。如果客户端出了问题，服务器模拟不应该受到影响。服务器在规定的时间内应该从客户端收到消息。如果一个客户端已经死机而不能满足这一要求，服务器可以忽略出了问题的客户端。这样的

客户端需要重新和服务器同步才可以继续游戏。

6.6.6 何时应使用多时管理

什么时候使用什么样的时间和同步管理取决于许多因素，包括网络构架和游戏设计。表 6.6.3 列出几种 MMROPG 和适用的时间管理方式。

表 6.6.3 MMORPG 构架和所适用的时间管理方式

	客户端进行预测，并给玩家的输入/指令标上时间戳	服务器给玩家的输入/指令标上时间戳
单一服务器	服务器回滚 延迟补偿 [Bernier00]	不需要特殊的时间管理
多服务器	多时间管理	服务器采用保守的时间管理和同步管理

6.6.7 总结

这篇文章论述了如何解决多服务器中存在的时间和同步管理。所给出的方法可以解决游戏状态同步和游戏响应时间之间的矛盾。

6.6.8 致谢

作者感谢 Dr. Richard Fijimoto 所给出的建议和对文章所进行的改动。

6.6.9 参考文献

[Aronson97] Aronson, “Dead Reckoning: Latency Hiding for Networked Games,” Gamasutra, 1997, available online at http://www.gamasutra.com/features/19970919/aronson_01.htm.

[Bernier09] Bernier, “Latency Compensating Methods in Client/Server In-game Protocol Design and Optimization,” in the 2001 Game Developer Conference Proceedings, available online at <http://www.gdconf.com/archives/2001/bernier.doc>.

[Bettner01] Bettner, Terrano, “1,500 Archers on a 28.8: Network Programming in Age of Empires and Beyond,” in the 2001 Game Developer Conference Proceedings, available online at http://www.gdconf.com/archives/2001/terrano_1500arch.doc.

[Chandy78] Chandy, Misra, “Distributed Simulation: A Case Study in Design and Verification of Distributed Programs,” IEEE Transactions on Software Engineering, SE-5(5):440-452, 1978.

[Cronin02] Cronin, Filstrup, Kurc, “A Distributed Multiplayer Game Server System,” in the Proceedings of the first workshop on Network and System Support for Games, Germany, 2002, available online at www.eecs.umich.edu/~bfilstru/quakefinal.pdf.

[DIS94] DIS Steering Committee, “The DIS Vision: A Map to the Future of Distributed Simulation,” Institute for Simulation and Training, 1994.

[Fujimoto98] Fujimoto, "Time Management in the High Level architecture," Simulation, Vol. 71, No. 6, pp. 388-400, December 1998, available online at http://www.cc.gatech.edu/computing/pads/PAPERS/Time_mgmt_High_Level_Arch.pdf.

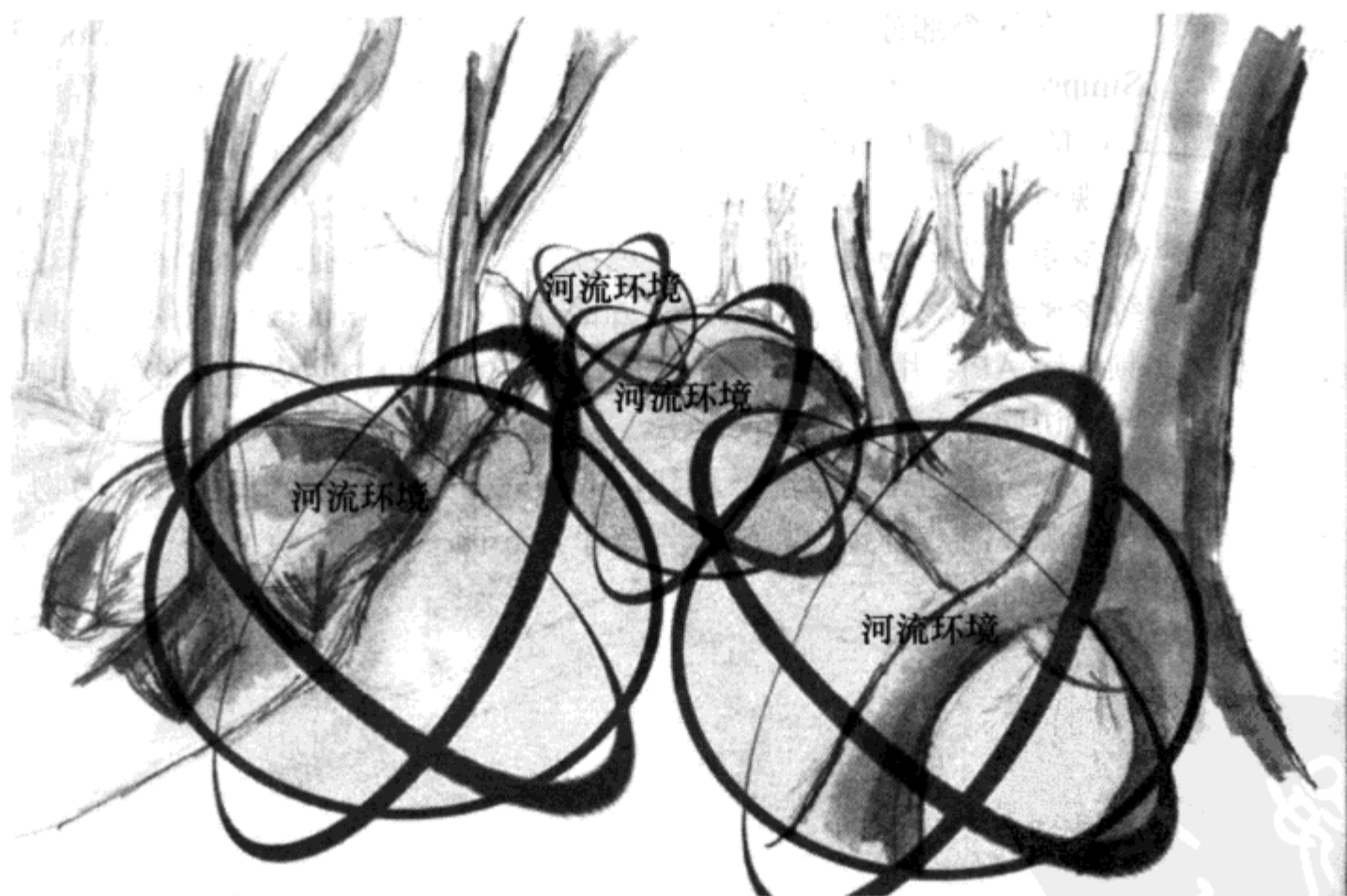
[Mauve02] Mauve, "How to Keep a Dead Man from Shooting," in the Proceedings of 7th International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services, October 2002, available online at <http://www.informatik.uni-mannheim.de/informatik/pi4/publications/library/Mauve2000a.pdf>.

[Mills92] Mills, "Network Time Protocol (version 3) Specification, Implementation and Analysis," RFC1305, March 1992, available online at <http://www.faqs.org/rfcs/rfc1305.html>.

[Ultima] Ultima Online, <http://www.uo.com>.



音 频



简介

作者: Eddie Edwards, Sony Computer Entertainment Europe

E-mail: eddie@tinyted.net

译者: 万太平

审校: 沙鹰

在过去十年中, 音频在游戏开发中似乎逐渐退居次要地位。我本人初次接触电脑游戏是在英国产的阿基米德计算机上 (Acorn Archimedes, 最早的 32 位 RISC 台式电脑)。在那个时候 (大约 1988 年), 把 20% 的 CPU 用于创建电脑游戏的音乐和音效是很正常的事情。而现在, 几乎所有的 CPU 时间都用在处理游戏的 AI、物理效果和图形图像上, 音频似乎已被打入冷宫——只有极少数例外。而且在游戏的创作过程中常常会忽略音乐和声效, 只在项目后期才进行补完。现在已经很少能找到一个程序员会称自己为“音频专家”了, 游戏音频这个领域似乎发展缓慢了些。可是, 我们还是能够从这个领域的专家那里收集到一些优秀的新技术。

在这个部分, 你会看到六篇关于游戏音频的文章。James Boer 和 Jake Simpson 描述了专门的编程技术——前者解释了一组用于音频程序开发的 C++ 的类, 而后者提出一种简单又很有效的方法, 在语音和人物动画之间实现自动“对口形” (lip-synching)。Borut Pfeifer 和 Frank Luchs 则探讨了更多系统范围的问题——创建声音脚本系统, 并把声音合成和游戏物理系统紧密结合。最后, Scott Velasquez 和 Joe Valenzuela 分别谈到了具体的音频程序 API: 用于环境音效的 EAX 和 ZoomFX, 以及可用来开放地、跨平台地访问声音硬件的 OpenAL。

我希望这些文章能鼓舞我们的读者。让我们投入更多的人力物力和开发资源在游戏音频上, 让音频程序设计再一次星火燎原!



7.1 OpenAL 简介

作者: Joe Valenzuela, Treyarch

E-mail: jvalenzu@infinite-monkeys.org

译者: 万太平

审校: 沙鹰

OpenAL (全称 Open Audio Library) 是专门负责 3D 定位音效方面的 API 接口库。与那些今日在游戏中得到普遍运用的较大的面向对象的库相比, OpenAL 是一个简单明了的替代方案。本文旨在简要介绍这个小型库。

OpenAL 简史

起初 OpenAL 并不正式, 它的第一个版本除了一个头文件和一个邮件列表外就没什么了。随着天才程序员 Terry Sikes、Jonathan Blow 和 Sean Palmer 等人不断注入其辛勤努力和远见卓识, OpenAL 开始形成其独特的工作重心: 通过一个作为那些新式图形库的补充的 API, 实现音频在三维空间的定位 (spatialized audio)。

OpenAL 的早期设计在很大程度上受到 OpenGL 的影响[Kreimeier02]。影响体现在方方面面, 从强调简单性直到最基本的命名空间 (namespace) 的正交使用 (orthogonal use): 也就是说, 若 OpenGL 用某方法做某事, OpenAL 也将照办。OpenAL 的早期演进过程中有一个重要的事实, 即 OpenAL 刚有第一个实现版本的时候就被应用于商业化应用程序的开发。这使得 OpenAL 能立即收到应用程序开发者的反馈, 因而早期开发简直是白热化。当然, 早期的狂热也不是完全没有问题, 下面就是证据: 用于指定一个循环音源 (Looping source) 的标记曾在一年之内做过 4 次修改!

最初设计的特色之一是其实现的简单性, 这在今天也被作为审核所有那些被提议增加的功能和扩展的标准。这是一个所有那些要加入库核心部分的功能都必须满足的极高标准。因而, 应用程序 (而非 OpenAL) 将负责大多数游戏中的场景管理。因为对于大部分 3D 音频应用程序, 其图形部分已经具备复杂场景的管理能力。事实证明这个并不是一个累赘的要求, 而是出于简化 OpenAL 库的需要。

7.1.1 OpenAL API

这部分将主要介绍 OpenAL 的接口, 从基本的概念到可选的扩展都会做一些说明。

1. OpenAL 入门

有一个笑话，说火柴盒的背面那么丁点地方就可以写得下那些必要的 OpenAL 库函数。这和事实相去不远，这充分体现了“简单事简单做，困难事别人做！”的精神。虽然新手不用写许多代码就变得能干起来，但良好地理解 OpenAL API 中的词汇，将对避免初期的混淆起很大的帮助。

OpenAL 从本质上讲是一个音频场景图库 (audio scene graph library)。它描述对象之间的一系列关系。大部分的对象体现了离散的概念。其中重要概念有设备 (Device)、渲染上下文环境的描述表 (context)、听众 (listener)、音源 (source)、缓冲器 (buffer) 等。大部分的 OpenAL 条目都和这些类型对象的创建、销毁或者属性改变有关 [Kreimeier02]。

一般而言，对象之间有如下关系：设备是最终输出 PCM (Pulse Code Modulation, 脉冲编码调制) 数据的硬件。一个 listener 属于且仅属于一个 context，而每个 context 也刚好只能有一个 listener。因此 context 就是在场景中聆听声音的对象实例。通常，每个场景中有一个 listener，有对应的位置和其他应用程序用户属性。缓冲器中存储的是原始 PCM 样本数据，不能直接播放。只有把缓冲器和音源关联起来，并播放该音源，声音才能被渲染出来。一个音源可以和多个缓冲器关联，此时我们称其拥有一个缓冲器队列 (Buffer Queue)。

音源和缓冲器一般通过名字 (name) 来引用，名字是整形标识符 (不同的对象类型具有唯一对应的名字)。例如，没有两个音源名字会相同，但它们可能与某些缓冲器的数字 ID 重复。

对象初始化以及名字绑定的语法是 `alGen{Object}`。相应地，销毁对象时调用 `alDelete{Object}`。例如，分别调用函数 `alGenSources()` 和 `alDeleteSources()` 来创建和销毁音源对象。创建 context 和 device 的函数调用与此不同，稍后我们会详细讨论。

音源是和 context 相关的。在一个 context 内有效的音源的名字在其他的 context 中无效。缓冲器是和 context 无关的，创建缓冲器无需引用任何当前活动的 context。缓冲器能够同时在多个 context 中与多个音源关联。

这些对象中的大部分都具有一些可以直接设定和查询的属性 (Attribute)。属性有一个特定的类型，也有默认值。最常用的是音源属性，通过音源属性可以使缓冲器和一些音源关联，还可以设定某一音源的位置等等。listener 与音源在设定和查询属性方面具有相似的语法，都是 `al{Object}{n}{if}{v}`。假如你已经习惯了阅读 OpenGL 文档，对于这样的语法你会也很熟悉。大部分属性访问都是以数字 (n) 或者向量 (v) 形式来表示，而所传递或接受的参数的类型是这样指定的：i 代表整数，f 则代表浮点数。例如，音源的位置通过函数 `alSource3f()` 或 `alSourcefv()` 带上 `AL_POSITION` 标记来设置。

最重要的缓冲器属性，即组成声音的 PCM 样本集，是通过函数 `alBufferData()` 来设定的。

下面是一小段 OpenAL 程序例子。

```
// 打开设备，创建设备
ALCdevice *dev = alcOpenDevice(NULL);
ALCcontext *cc = alcCreateContext(dev, NULL);

alcMakeContextCurrent(cc);

// 创建音源和缓冲器
```

```
ALuint bid, sid;
alGenSources( 1, &sid );
alGenBuffers( 1, &bid );

// 取得 pcm 数据, 用缓冲区来关联它
ALvoid *data;
ALsizei size, bits, freq;
ALenum format;
ALboolean loop;

alutLoadWAVFile("boom.wav", &format, &data, &size,
&freq, &loop);
alBufferData(bid, format, data, size, freq);

// 用音源关联缓冲器
alSourcei( sid, AL_BUFFER, bid );

// 播放音源然后等待直到完成
// 然后摧毁它
alSourcePlay(sid);
ALint state;
do {
    alGetSourcei(sid, AL_SOURCE_STATE, &state);
} while(state == AL_PLAYING);

alDeleteSources(1, &sid);
alDeleteBuffers(1, &bid);

alcMakeContextCurrent(NULL);
alcDestroyContext(cc);
alcCloseDevice(dev);
```

如上述程序所示, 函数 `alcOpenDevice()` 用于打开设备, 它带有一个可选的设备指示字符串参数。该字符串参数的语法和含义是与实现相关的。这意味着允许应用程序指定另外的后端或与设备相关的配置参数。在 GNU/Linux 的参考实现中, 该设备字符串参数作为 LISP 语言风格的标记被解释 (interpret), 能够指定多个后端以及一些属性诸如采样率和后端相关功能。

函数 `alcCreateContext()` 用于创建渲染上下文环境, 创建时需要指定一个设备用作 context 中混音的渲染目标。另外这个函数还可以带可选的 context 属性表参数, 形式是以零终止的整数对。需要由实现支持的 context 属性包括有 `ALC_SYNC`、`ALC_REFRESH` 及 `ALC_FREQUENCY`。`ALC_FREQUENCY` 和 `ALC_REFRESH` 会影响 context 渲染的性能与保真度, 而 `ALC_SYNC` 令 context 仅在调用函数 `alcProcessContext()` 进行更新后才会进行混音。因为可能有多个 context, 所以需要用函数 `alcMakeContextCurrent()` 来指定当前的 context。

如前例显示的那样, OpenAL 在语法、编码风格和习惯上都刻意模仿 OpenGL。做出这个决定是为了在一定程度上迎合那些已经熟悉 OpenGL 这个流行图形库的开发人员, 也是为了仿效 OpenGL ARB 所代表的切合实际的设计原则。

2. 进阶

可通过 `alSource{n}{if}{v}` 条目来设定音源的属性。

音源属性可以分成三组：第一组影响音源在 OpenAL 世界中的物理位置，例如 `AL_POSITION` 和 `AL_VELOCITY`；第二组表示“旋钮和转盘”如何影响音源的播放，例如 `AL_PITCH`；最后一组则是对音源的高级别管理很有用处的状态属性，例如 `AL_LOOPING` 和 `AL_SOURCE_STATE` [Kreimeier02]。

使用 `AL_POSITION` 属性来设置的音源位置是世界坐标系中的位置。只有带了附加属性 `AL_SOURCE_RELATIVE` 的时候才例外，这个属性告诉实现程序以渲染上下文环境的听众作为它的原点来定位。这个属性在像类似可能从头盔发出头部相关的声音或者像音乐这种“2D”声音来说很有用处。对于那些无需定位的声音，常常使用 `internalFormat`（多通道）扩展来实现，这会在后面进行介绍。

音源的 `AL_PITCH` 属性用于控制某一声音的相对音高。取值为 1.0 的时候，渲染的音源上无需调高。每减少 50% 会导致一个八度（-12 半音）的音高变化 [Kreimeier02]。在 GNU/Linux 实现下，多普勒频率滤波器计算多普勒效应作为现有的音高属性的放缩因子。在应用程序中生动地使用可以达到非常好的效果。而通过软件实现音源的音高变化的代价是昂贵的，因此使用前合理地判断是必要的。

多普勒效应说明了 OpenAL API 的一些亮点。假如想使用多普勒效应，则必须设定 listener 和音源的一些属性。

```
ALfloat l_pos[] = { 0, 0, 5 };
ALfloat s_pos[] = { 0, 0, -5 }, s_vel[] = { 0, 0, 1 };
ALfloat zeros[] = { 0, 0, 0 };

alListenerfv(AL_POSITION, l_pos);
alListenerfv(AL_VELOCITY, zeros);

alSourcefv(sid, AL_POSITION, s_pos);
alSourcefv(sid, AL_VELOCITY, s_vel);

alSourcePlay(sid);
ALint state;
do {
    s_vel[2] += 0.001;
    s_pos[2] += 0.001;

    alSourcefv(sid, AL_VELOCITY, s_vel);
    alSourcefv(sid, AL_POSITION, s_pos);

    alGetSourcei(sid, AL_SOURCE_STATE, &state);
} while (state != AL_PLAYING);
```

本例中需要注意的是，音源位置的计算不是推导出来的——而是由应用程序明确设置的。同时假定所有的位置和速度都是即时的。

OpenAL 通过缓冲器排队机制支持声音的流式播放。缓冲器排队是多个缓冲器与单一音源相关联的一种机制。当音源播放时，连续对各个缓冲器进行渲染，就好像这些缓冲器组成

了一个连续的声音。这可以通过一些特殊函数来控制。

流音源的工作一般是这样的。音源里的一批缓冲器通过 `alSourceQueueBuffers()` 函数进行排队，然后播放音源，接下来用属性 `AL_BUFFERS_PROCESSED` 来查询。该属性得出已经处理好的缓冲器的数量，从而允许应用程序使用 `alSourceUnqueueBuffers()` 函数删除那些已经处理好的缓冲器。`alSourceUnqueueBuffers()` 函数将从队列头部开始依次将处理好的缓冲器删除。最后，其余的缓冲器在音源上排队 [Creative03]。当缓冲器正在播放时，试图移去缓冲器会得到一个错误。

```
// 使用排队机制来关联到缓冲器第一个集
alSourceQueueBuffers(sid, NUMBUFFERS, Buffers);

alSourcePlay(sid);

ALuint count = 0;
ALuint buffers_returned = 0;
ALint processed = 0;
ALboolean bFinished = AL_FALSE;
ALuint buffers_in_queue = NUMBUFFERS;

while (!bFinished)
{
    // 取得状态
    alGetSourceiv(sid, AL_BUFFER_PROCESSED, &processed);

    // 假如播放完毕了一些缓冲器，然后让它们退出队列
    // 然后装载新的音频，再把它们装入队列
    if (processed > 0)
    {
        buffers_returned += processed;

        while(processed)
        {
            ALuint bid;
            alSourceUnqueueBuffers(sid, 1, &bid);

            if(!bFinished)
            {
                DataToRead = (DataSize > BSIZE) ? BSIZE : DataSize;
                if (DataToRead == DataSize)
                    bFinished = AL_TRUE;

                // 从音源总读出 Data To Read 字节的代码省略
                // ...

                DataSize -= DataToRead;

                if (bFinished == AL_TRUE)
                    memset(data + DataToRead, 0, BSIZE - DataToRead);

                alBufferData (bid, format, data, DataToRead, wave.SamplesPerSec);
            }
        }
    }
}
```

```

        // 对缓冲器排队
        alSourceQueueBuffers(sid, 1, &bid);
        processed--;
    }
    else
    {
        processed--;
        if (buffers_in_queue-- == 0)
        {
            bFinished = AL_TRUE;
            break;
        }
    }
}
}
}

```

3. 空间定位

OpenAL 的核心是将声音的衰减表现为某一距离函数。OpenAL 有一系列的距离模型可以在运行的时候选择，不同模型的 Direct3D 兼容性不同，应用程序支持的容易程度不同，与物理公式的一致性也有所不同。

函数 `alDistanceModel()` 用于在不同的距离模型中进行选择。默认的距离模型是 `AL_INVERSE_DISTANCE`，遵守下面的公式：

$$G_{db} = \text{clamp}(GAIN - 20 \times \log_{10}(1 + Rf \times (dist - Rd) / Rd), MinG, MaxG)$$

此公式中 `Rf` 和 `Rd` 对应于音源的两个属性：`AL_ROLLOFF_FACTOR` 和 `AL_REFERENCE_DISTANCE`。`MinG` 和 `MaxG` 分别对应于音源的最小增益属性 `AL_MIN_GAIN` 和最大增益属性 `AL_MAX_GAIN`。参考距离 `dist` 是 `listen` 体验增益 (`GAIN`) 的距离。依音源而定的 rolloff 系数 (高低频规律性衰减系数) 能够在值变化量的负方向上改变音源的范围。当 rolloff 系数为 0 表明对于音源没有衰减 [Kreimeier02]。

OpenAL 也提供和 IASIG I3DL2^① 兼容的距离模型。使用 DS3D^② 的用户会更熟悉此模型。该距离模型可表示为如下公式表示为：

$$G_{db} = \text{clamp}(GAIN - 20 \times \log_{10}(1 + Rf \times \text{clamp}(dist, Rd, Md) / Rd), MinG, MaxG)$$

这个公式里增加的是，把音源的距离限定在参考距离和音源特定的最大距离 `AL_MAX_DISTANCE` 之间。其他考虑中的距离模型是简化的线性模型，允许增益降低至 0。

4. 扩展 (Extension) 与 alut 库

OpenAL 具有和 OpenGL 相似的可扩展性。应用程序首先调用函数 `alGetString (AL_EXTENSIONS)` 来询问实现。此函数返回一个可在其中搜索特定标识的扩展字符串。

① 译注：IASIG I3DL2 是交互音频工作组所制定的一项 3D 定位音效指导准则，全称是 Interactive Audio Special Interest Group; Interactive 3D Audio Rendering Guidelines - Level 2。

② 译注：微软的 DirectSound3D 的简称。

此外,函数 `allExtensionsPresent()` 可以确定是否存在某个扩展。一旦确定某一扩展的存在,应用程序将能够通过函数 `alGetProcAddress()` 和 `alGetEnumValue()` 取得特定的函数和枚举标记。

扩展依赖于实现,它们的存在或必然性是非常不可靠的。在 GNU/Linux 实现中最为常用的扩展是 Ogg Vorbis 和 MP3 扩展,分别提供了一些用于播放压缩的文件格式的函数。同样重要的是一个提供四声道 (Quadraphonic) 立体声的扩展,以及允许使用多通道音频采样格式 (如立体声音源) 的 `internalFormat` 扩展。

Creative 的实现中最常用的就是允许控制 EAX 属性的 EAX 函数集。EAX 用来向核心库中增加高级特性,包括 listener 和个别音源之间的混响 (reverberation)、反射 (reflection)、封闭 (occlusion) 等。Creative Labs 公司的 Garin Hiebert 特别提到最新版本 EAX 的一些问题“包含了更多控制……音效在 listener 周围发生左右漂移,根据 listener 与音源之间尺寸可变的间隙进行滤波”等等。EAX 属性用 `EAXGet()` 和 `EAXSet()` 来操作。两者格式相同:

```
ALenum EAX{Get,Set}(const struct* propertySetID,
ALuint property, ALuint source,
ALvoid* value, ALuint size);
```

下面是一个设置 EAX 属性的例子:

```
// 省略了分配音源和缓冲器及设置它们的属性的代码
ALuint Env = EAX_ENVIRONMENT_HANGAR;
eaxSet(&DSPPROPERTYID_EAX20_ListenerProperties,
      DSPROPERTY_EAXLISTENER_ENVIRONMENT |
      DSPROPERTY_EAXLISTENER_DEFERRED, NULL, &Env,
      sizeof(ALuint));
```

一般而言, EAX 提供一系列面向室内的效果,能够使应用程序增加真实感 (若实际上并未进行物理建模) [EAX02]。

OpenAL 核心库中没有用于处理文件格式的函数。该项功能由 `alut` 辅助库提供实现。函数 `alutLoadWavFile()` 和 `alutLoadWavMemory()` 可以加载不同版本的 WAV 文件格式。在提供便于应用程序载入音频文件函数的同时, `alut` 还有简化初始化和结束程序的 `alutInit` 和 `alutExit` 例程。它们隐藏了 context 和设备的初始化细节,不过要稍稍损失一些灵活性作为代价。

7.1.2 有关 OpenAL 的实现

因为 OpenAL 是一个规范,而不是一个实现,用户必须注意在不同的实现之间有何差异。如果没有理解多数实现的原理,很可能就会损失音质或性能。

1. 常犯错误

OpenAL 的新用户往往会碰到相同的一些问题。本节将描述最常见的典型问题。

2. 多通道音频

通过 `alBufferData()` 函数与一个缓冲器名相关联的数据由库进行复制，假如必要的话还可以转换为某个更加合适的格式。这里没有定义实现应选择的确切格式。因为不同的后端或者驱动程序对于各自的数据格式都有特殊的需求，因此这里要迁就性能。大部分空间定位的声音来自于和独立于通道的效果的应用。请记住这一点，那么多通道音频就很明显是代表“预先在空间定位的”声音，而用于定位音频的目的不清楚。在游戏中，多通道音频主要用于音乐，因为在这种情况下音源的定位可被忽略，音源是作为一个环境音效来播放的。

然而，因为应用程序并不了解实现的内部格式，当实现端使用单通道的格式时，这就冒了可能丢失多通道音质的风险。为了巧妙地克服这个问题，一些实现提供 `internalFormat` 扩展，它可以向实现发出指示，应当保留数据的多通道属性。该扩展提供一个额外的函数：

```
alBufferData_LOKI(ALuint buffer, AEnum format,
    ALvoid *data, ALsizei size,
    ALsizei freq, AEnum internalFormat);
```

此函数和 `alBufferData()` 有相同的语法和语意，但是附加的 `internalFormat` 参数除外 [Kreimeier02]。

3. 环境音效

没有任何属性来告诉驱动程序某个音源是环境的，更不用说是定位的。为了创建一个环境音效，应用程序应当通过使用 `AL_SOURCE_RELATIVE` 属性来保证音效的方向正确，必要时一同使用 `internalFormat` 扩展。

4. 枚举标记的值

枚举标记的值，除了 `AL_TRUE` 和 `AL_FALSE` 以外，在跨实现之间不保证有相同的值。对于不希望改动重新编译的驱动程序的情况，应用程序应通过 `alGetEnumValue()` 查询所需标记的值。

5. 常见的性能问题

除了一致性和保真度问题之外，新手们通常会碰到性能问题。特别在软件实现的场合表现得更加明显（例如 GNU/Linux 的参考实现），但任何应用程序在音频处理方面运算能力的预算都有限。虽然越来越多的处理正被移植到硬件上，但很多复杂的效果仍依赖 CPU 计算 [Kreimeier01]，即使是硬件加速的实现也是如此。因为这个原因，与软件实现有关的一些常见问题对于所有驱动程序或多或少都是适用的。

正如你可能担心的那样，有越多要处理的数据，就需要越多的 CPU 时间来处理。应用程序应该总是使用最低可以接受的采样率。为了减少内部转换，设备应该和当前渲染的 `context` 保持一致的采样率。这是函数 `alcOpenDevice()` 中与实现相关的设备标识符参数的常用场合。在 GNU/Linux 实现中的相应代码如下：


```
int attrlist[] = { ALC_FREQUENCY, 22050, 0 };
ALCdevice* dev = alcOpenDevice("((sampling-rate 22050))");
ALCcontext* cc = alcCreateContext(dev, attrlist);

alcMakeContextCurrent(cc);
```

在音源上计算最复杂的滤波器是那些影响音高的滤波器。在 GNU/Linux 实现中，就是 AL_PITCH 和多普勒频率滤波器。使用 1.0 的声高值使应用程序关闭音高的处理。应用程序通过以参数 0 调用函数 `alDopplerFactor()`，可以关闭多普勒处理。但若想在个别的音源上关闭多普勒效果没有正式的方法，应用程序若想这么做只有通过 AL_SOURCE_RELATIVE 将音源相对 listener 的速度设定为 0。

7.1.3 实现一致性指南

那些雄心勃勃的程序员总是打算跨多个平台和多个实现兼容。但当他们发现必须将大量的时间花费在区分 bug 和实现相关的行为上时，热情就被浇上了一瓢冷水。为了减少观念混乱和倡导程序之间的互用性，那许多无必要地异于其他实现或异于 1.0 版标准的领域，将会整合到一起。这可以通过下面几步达到：

- 共享更多的代码和通用枚举标记的值；
- 增加部分扬声器的摆放功能；
- 标准化的 OpenAL 扩展。

1. 共享更多的代码和通用枚举标记的值

有大量的代码（即使是在硬件加速的实现中）用于处理音频库中的一些传统功能。有些任务，比如记录对象、加载音频文件及不同内部文件格式之间的转换，在所有的平台上都是类似的。任何开发人员都会得出相近的结果。而 Creative 公司和 GNU/Linux 实现中宽容的许可证条款旨在鼓励代码的重用。

如前面提到的，在 OpenAL 1.0 中没有定义枚举标记的值。尽管在实现之间对于二进制的兼容性没有任何需求，但一些很小的改变就可提高使用的便利性。

2. 增加部分扬声器的摆放功能

在 OpenAL API 中没有用于扬声器摆放的接口，这是因为扬声器摆放的配置者是最终用户，而非应用程序开发员，因此扬声器的摆放最好在运行环境下处理。声卡厂商通常把配套软件打包，允许用户指定扬声器的摆放。虽然不太可能在 OpenAL 核心库中增加关于扬声器配置的综合 API，但已计划增加一些作为补充的操作系统功能机制。

3. 标准化的 OpenAL 扩展

众多平台上现存的许多扩展最终会或被植入核心的 AL 部分，或在一个辅助库中找到安生立命之所。越来越多的实现已经采用了用于加载压缩音频格式文件的扩展，及有关额外的距离模型的扩展。在未来，共享共同的扩展必将带来更加一致的代码。

7.1.4 未来 OpenAL 的发展蓝图

借用 Woody Allen 的话来说, 函数库的开发就像是一条鲨鱼, 假如停止了前进, 它就会消亡。OpenAL 持续维持其良好的状态, 有赖于对未来发展的计划, 紧跟技术发展的脚步, 以及同行的贡献。若缺少了组织的成长, 开发的努力就会落空。若没有技术的进步, 专业人员的兴趣也会萎缩。而若是缺少了同行的贡献, OpenAL 就无法和 DirectSound 这样的工业标准相竞争。

未来计划组建一个技术指导委员会 (ARB, Architectural Review Board) 类型的团体来组织 OpenAL 的开发。以前, 这个角色都是由 Loki 和 Creative 公司中的少数几个高手义务担当的。在未来, 库的结构和组成都必须通过这一专业组织来决定 (或者定型)。具体地说, ARB 不但将决定一般意义的 API 趋势, 也将决定核心库中需要哪些特征, 并判断什么行为可被标准化, 什么又必须留给实现来决定。作为榜样, OpenGL ARB 提供了一个很好的基础, 尽管需要不断地集成新的功能, 但只需做很少的主要改动。这个成就真的不小。

然而, 假如没有值得倡导的技术, 这个组织又有什么用呢? OpenAL 被设计成希望能够完整地取代那些过于关注 API 本身而非音频的音频 API。因为它的接口简单但完整, 对于那些既想做 3D 定位音频又想跨平台的音频程序员来说, 是个不错的选择。

OpenAL 和其他音频库的真正区别就是你的参与。一直以来, OpenAL 的用户们贡献了大量源代码, 帮助编写辅助文档, 并在非正式的论坛上提供支持。OpenAL 不但受益于使用 OpenAL 的开发者, 更需要他们积极参与自身的开发工作。

对于 OpenAL 的更多信息可以访问 www.openal.org 或 <http://developer.creative.com> 中的 OpenAL 部分。这些站点有从安装、使用, 到发布 OpenAL 的信息, 同时还有邮件列表和怎样贡献的说明书。加入 OpenAL 吧, 它就是你需要的音频库。

7.1.5 总结

OpenAL 是一个跨平台的音频接口, 希望对音频开发起到“OpenGL”对图形学所起的同样作用。随着 OpenAL 的接口继续发展, 音效开发人员无需花太多的时间考虑硬件问题, 而是将更多的时间用于改进实际的游戏音效。

7.1.6 参考文献

[Creative03] Creative Labs, *OpenAL Programmers Guide*, available online at <http://developer.creative.com>.

[EAX02] Creative Labs, *Environmental Audio Extensions: EAX 2.0*, available online at <http://developer.creative.com>.

[Kreimeier01] Kreimeier, Bernd, “The Story of OpenAL,” *Linux Journal* (January 2001): pp. 102–107.

[Kreimeier02] Kreimeier, Bernd, et al., *OpenAL Specification and Reference, Snapshot*, available online at www.openal.org.

7.2 简单的实时 Lip-Synching 系统

作者: Jake Simpson, Maxis

E-mail: jmsimpson@maxis.com

译者: 万太平

审校: 许竹钧

当你说话时,唇和舌形成某种构造,当气流通过就发出声音。声音的最小单位是音素(Phoneme)。在英语中大约有 40 个音素——包括元音(vowel)、鼻音(nasal)、边音(approximant)和爆破音(plosive)。因为还有许多地方口音和方言特有的音素,所以 40 只是个约数。

我们的简单对口形系统并非想要进行音素级别的模拟。要想进行精确模拟的话,有多种价格昂贵的商业化系统可供选用,它们使用的技术各有千秋,得到的声音的音质也各有高下。

读唇者(lip reader)通过反复练习音素学会了读唇。但绝大多数未经专门训练的人并不能(在意识层面)意识到某个嘴形应该对应什么声音。然而,若声音产生的时机和嘴形变化的时机不协调,则普通人也能发觉。例如,看电影时候看到声音和画面不同步,我们会立刻发觉。也就是说,我们的大脑可以近乎本能地“看穿”声音和嘴唇动作之间的不匹配。

我们这个简单的对口形系统就是利用这个原理。我们并不打算去精确地模拟所有音素——那样做难度过高因而有时并不适用。我们只要保证嘴唇能够适时的动就行了,也就是嘴唇会随着声音的产生而同步动作。这个对口形系统已经在几款已经上市的游戏中得到应用,每一款都得到了“这口形对得真好”的赞美,可见玩家并不能觉察出我们并没有在软件内部精确地“对口形”!

7.2.1 实现

用最简单的话来说,我们所要做的就是逐帧地计算距离当前“播放位置”一定范围内音效样本振幅的平均值。然后,用这结果来决定应该显示哪一套预先设定的嘴形。

更明确地说,在你播放一个音频样本的时候,对于每一帧都预测未来 100 毫秒之间的样本。然后对于这些样本取它们振幅的平均值来决定你使用哪个合适的嘴型。样本的某些部分平均振幅为 0,保持沉默,等同于闭着的嘴型。(假如这个值为-1 表明样本已经播放完毕,所以此时面容应该开始恢复平静)

这个系统是否有效，高度依赖于负责建模的美工（或者动画师，或者材质方面的美工），因为是他们向你提供多种彼此之间差别很大的嘴型，因而你可以在不同的嘴形之间插值，而制造出嘴唇运动的感觉。

为了保证这个系统的不错效果，要求帧速率必须比较高才行。但任何对口形的解决方案都需要比较不错的帧速率，所以这是肯定的。对于那些重新改变振幅值的嘴形速率是可变的，主要是看起来感觉舒服与否。通常，你需要有足够的时间用于混合新的位置动画，这样眼睛才能轻松地辨别。如果嘴形改变过快，混和的效果看起来有点不可思议，好像嘴唇飞起来一样；假如过慢，可能会使得嘴原本应张开（或者合拢）的时候却刚好处在相反的位置，这样就不合我们的要求。过去的经验表明，取样率为 10Hz 或者稍微小点最佳。这通常需要你不断地调节，直到满意为止。

假如你能够使用高取样频率的声音，就不必对这个音量近似值读取每个样本，你可以隔一个读一个，或者每三个读一个，无论哪种方法都能产生很好的效果。

7.2.2 动画方面需要注意的事项

要真正实现一个最有效的对口形系统，方法之一就是对你取得的口形数据进行处理。实际上，播放动画或者纹理需要一些小技巧。

有些游戏根本不重新计算模型，而只是使用另一张嘴形不同、或干脆闭合的唇的纹理贴图来替换当前纹理。这种方法也可能产生令人惊讶的效果——Raven Software 公司的游戏《星际迷航：精英力量》（*Star Trek Voyager: Elite Force*）就是一个很好的例子。无论采用何种方法，都需要小心谨慎以确保只影响嘴部的纹理；假如你对于整个脸部作处理，就不能做脸上部表情和说话动画的混合。

如果你希望调整模型的网格，那么请记住 Ritual Entertainment 公司的《重金属 2》（*F.A.K.K. 2*）的教训——不要仅仅只在下颌固定一块骨骼然后上下摆动。真正的人体是不会这样动作的，这样做只会搞得每个角色都像提线木偶那样。在我们目前正在开发的一个游戏中，我们对一个骨骼动画和渐变对象进行混合，效果非常让人满意。对于每一帧，我们使用一串嘴形，然后在它们之间做快速混合。

经验表明，平均说来，大约 5~6 个嘴形就能提供足够的多样性，能够用来产生看上去可以接受的嘴形动画了。图 7.2.1 中提供了嘴巴从完全张开到闭合的 5 个嘴形，以供参考。

假如美工能正确地建立脸部模型（在那些真正肌肉所在的位置放上渐变的对象），你就能够将脸上的“表情集合”混合到实际的讲话当中。角色就能眨眼、睁大、斜视，以增强言语的效果。将角色说话时的情绪表现出来，能起到很好的效果，假如你能做到，就应该去做。

在创建嘴形的时候，不能只绘制出一些大小不等的张嘴动作，这会使你的角色看上去只是在用大小不等的音量说“oooooooo”。努力去寻找一些特定的嘴形来匹配特定的音素，比如使用张得更大的口形来表示高音等。即使嘴巴的形状还不能完全和语音匹配，但问题也不大。我们追求的是全面的效果。

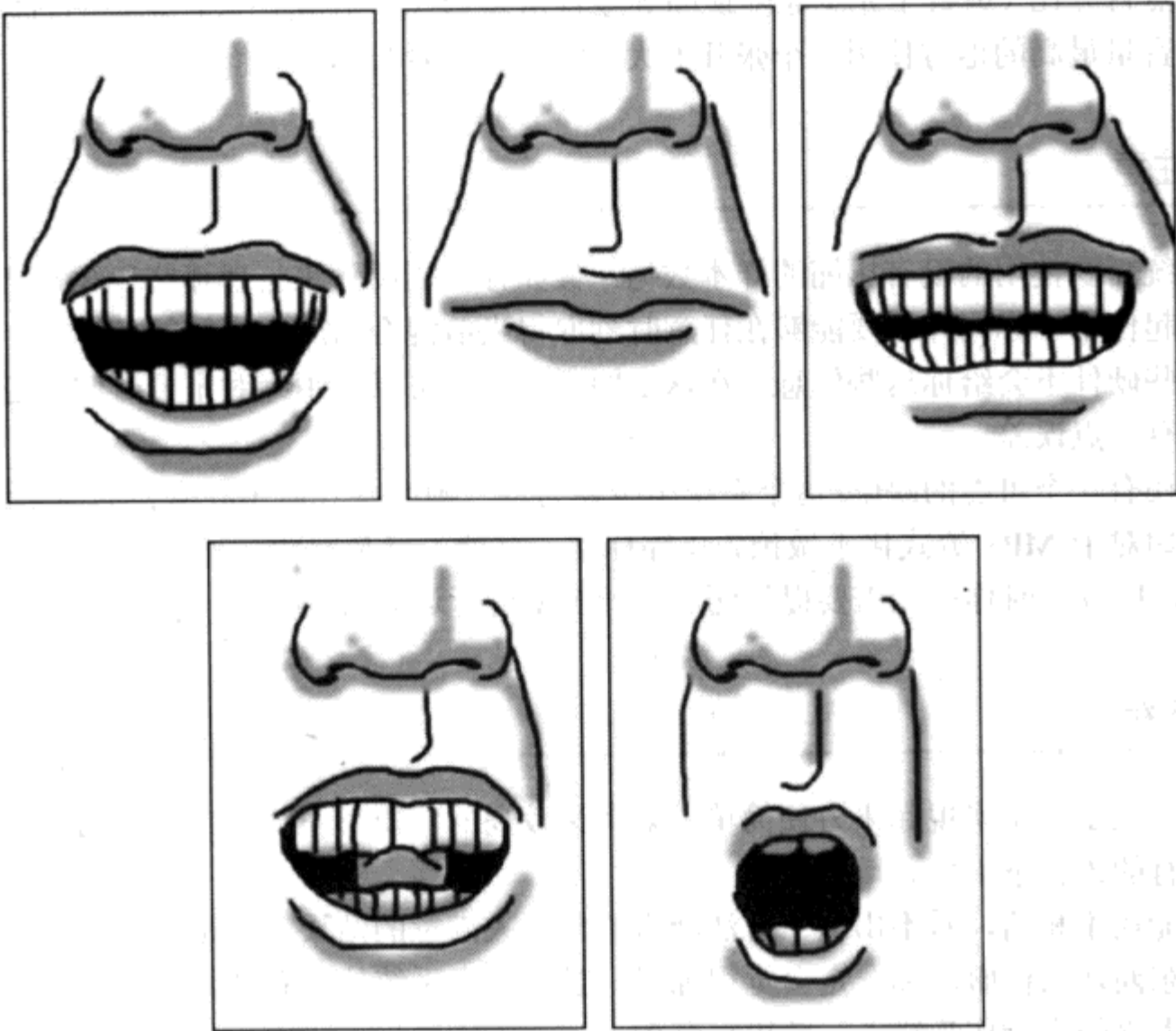


图 7.2.1 从合拢到完全张开的各种嘴形

7.2.3 声音音量的水印标记

假如你按照声音振幅的 12%、26%、47%、58%、80%来设定 5 个（随意）嘴吧的形状，那么问题出现了，47%对应哪一个呢？

部分讲话样本的音量无疑会比其他的要大。因此就可能出现，当角色低声私语时，因为平均音量太低，所以对口形系统始终都只在整个样本中挑选一样的嘴形。

一个可以修正这个缺点的办法是围绕语音样本循环，然后基于样本中最高音的百分比设定它们声音水印标记音量。对于任何指定的样本，你需要确定最大音值，这样就可以将实际的值分配到些百分比之上。在装载的时候可以完成这样的操作（对于流式处理的声音不是一个好主意，因为它会增加装载时间），或者你也可以将它看作一个预处理的步骤。

有人可能争辩说，把声音的音量水印百分比按照 20%、40%、60%等设定，将会工作得很良好，但是经验表明没有这个必要。假如讲话的样本是 80%，却落在 60%~80%的范围之内，那么相当数量的动画都会以相同形状的嘴形结束。在这里，所有的想法都有一个共同点，要防止相同的嘴形频繁出现。可能你会认为每个样本都应该重新生成百分比，但通常会做过头。要找到你认为最好的范围需要做一些试验。不要采用纯粹随机的值——可能看起来很好，但是需要很多调整来使得可能的随机范围变得正确。通常，这不值得花很多时间。

需要特别注意的是，假如声音样本比较安静，则最好检查最大的音量并使用稍微不同的

音量最高点百分比（或甚至完全删除最高等级音量）。毕竟，在角色低声说话的时候，若是在样本中音量最高的地方使用一个张开很大的嘴形，会看起来好奇怪。

7.2.4 注意

本系统需要使用的没有压缩的样本数据。假如你使用硬件来处理那些压缩数据，这个方法就不会起作用。你同样需要能够在任意时刻取得当前混合器所在指针的位置。当做内部混频时，有些硬件不会给你这些信息。在这个例子上，你可以使用绝对定位时间来避开这个问题，但会有一点误差。

另外还有一个可能的缺陷：这个系统至少要为每一帧准备其后 100 毫秒的原始波形数据。例如，假如对于 MP3 流式化不做预先压缩样本，这就可能是问题。另外，MP3 是被解压为 4kB 的块，所以有时你可能需要提供两个缓冲器来进行运算。

7.2.5 总结

本文介绍了一个实现起来方便简单，效率又较高的对口形系统，它最大的优点就是可以直接在所有的语言下工作。

你可能还不相信，看不出这个方法能否给出令人满意的结果。可是当你看到人们对眼前所看到的東西那么信服，你一定会觉得惊讶。尝试这个系统然后看能否为你所用。另外，尝试使用非人声的音频作为输入，效果很有趣。

感谢 BJ West 提供的嘴形图像。



7.3 动态变量和音频编程

作者: James Boer

E-mail: james.boer@gte.net

译者: 万太平

审校: 李鸣渤

在音频和音乐编程领域中, 在某段指定时间内把变量(例如一段正在播放的音乐的音量变量)从一个值逐渐调整到另外一个值, 是最常见的任务之一。虽然说诸如此类的简单插值实现起来不费吹灰之力, 但一再重复编写同样的代码还是会让人感到乏味。有一个方法可以解决这个问题, 那就是使用 C++ 的运算符重载, 设计一个能随时间计算自身插值的智能变量 [Boer02]。

7.3.1 动态变量是什么?

动态变量是 C++ 对象, 除了可以自动地在一段指定时间内在两个给定值之间进行插值计算之外, 它的用法和普通变量没什么区别(通常表示浮点数值)。动态变量的类非常简单, 但能够使你在集中计算复杂的插值设定时, 免去很多乏味重复的编码过程。因为 C++ 允许对基本运算符进行重载, 我们编写的类能够模仿 C++ 语言的内部数据类型的行为, 所以能获得更简洁、使用起来更直观的代码 [Meyers96]、[Meyers98]。

7.3.2 动态变量类

程序清单 7.3.1 展示了一个简单的动态变量类。

程序清单 7.3.1 动态变量 DynamicVar 类跟踪一个变量的值随着时间改变

```
class DynamicVar
{
public:
    DynamicVar()
    {
        m_fVar      = 0.0f;
        m_fTime      = 0.0f;
        m_fTimeTarget = 0.0f;
    }
};
```



```

void setVar(float fVal, float fTime)
{
    m_fTime      = 0.0f;
    m_fTarget     = fVal;
    m_fDelta      = m_fTarget - m_fVar;
    m_fTimeTarget = fTime;
}

void update(float fDeltaTime)
{
    m_fTime += fDeltaTime;

    if (hasReachedTarget())
        m_fVar = m_fTarget;
    else
        m_fVar += (m_fDeltaTime / m_fTimeTarget) * m_fDelta;
}

operator float()
{ return m_fVar; }

void operator = (float fVal)
{
    m_fVar      = fVal;
    m_fTarget   = fVal;
    m_fTime     = 0.0f;
    m_fTimeTarget = 0.0f;
}

bool hasReachedTarget()
{ return (m_fTime >= m_fTimeTarget); }

private:
    float  m_fVar;           // 当前值
    float  m_fTarget;       // 目标值
    float  m_fDelta;        // delta 值 (斜率)
    float  m_fTime;         // 当前时间
    float  m_fTimeTarget;   // 目标时间
};

```

DynamicVar 类的基本部分由下列这些运算符和函数组成：赋值运算符（Assignment Operator）用来设定对象所代表的“变量”的值；初始化函数（Initialization Function）告诉变量应如何在多少时间内进行插值计算；更新函数（Update Function）执行实际的插值工作；重载的浮点数类型转换函数负责从对象取得变量值；还有一个查询函数 hasReachedTarget() 判断插值是否已达到目标值。

7.3.3 在音频编程中使用动态变量



在音频编程中，可以利用简单插值操作的场合数不胜数。音乐流的基本淡入淡出过程就是一个简单的例子。播放一段模仿自然界音响的音乐（sound of organic origin）时，比方说播放一段表现呼号的风声的循环音乐时，可以随着时间流逝来改变音高（pitch）、音量（volume）和音调（pan）（对于 3D 声音还有位置），从而为玩家创造动态程度更高的体验。使用动态插值能够避免使用那些频繁重复播放的环境音轨而带来的问题，具体说也就是用户能轻易辨认出系统正在循环重复地播放声音而带来的问题。对于在环境音场中使用动态变量的例子，你可以研究一下附带 CD-ROM 中的 SoundscapeTest.exe 样例程序。

我们将给出一个更加有趣的动态变量应用——那就是使用与 DynamicVar 相同的原理和基本机制来实现音频包络线（Envelope，又称波封）控制类。

从图 7.3.1 能够看出音频包络线只不过是某段时间内一系列基本的线性插值。通常，持续时间是一个未知变量，其值将在运行时确定。例如任何能够在游戏中产生重复声音的物体，无论是汽车引擎、电梯、或任何其他能够被玩家实时控制的装置。但是，不论是事先编好的或实时控制的，我们都能很容易地创建控制类来处理。在我们示范的类中，传递一个正值到参数中，代表时间是固定的，而传递负值则代表持续时间需要在运行时加以控制。

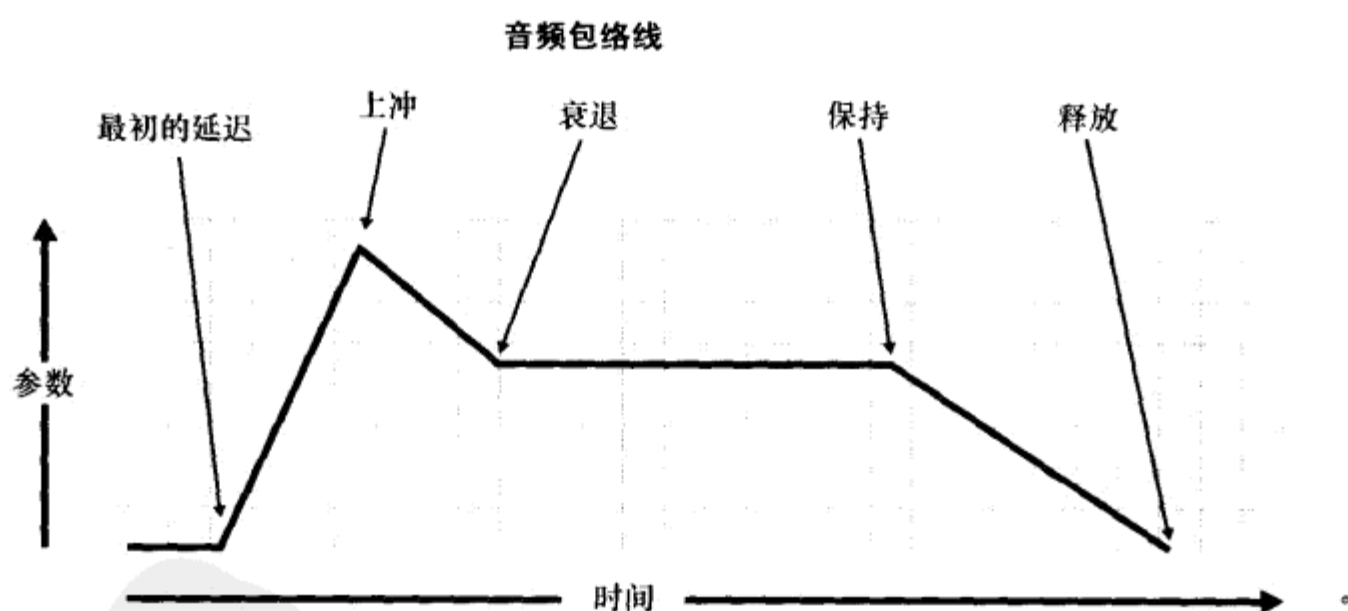


图 7.3.1 音频包络线

请记住，这个音频包络线控制不仅能表示音量，也可以表示诸如音高等其他参数。例如，电梯启动时音高由低变高，当运行到终点时音高则从高变低。

程序清单 7.3.2 给出了一个开发完成的音频包络线控制类 EnvelopeVar。EnvelopeVar 和 DynamicVar 很相似，设计思路是为了模仿单个变量随着时间流逝做插值运算，而这一次的插值运算更加复杂，所以需在内部使用一个 DynamicVar 来执行工作。

程序清单 7.3.2 基于 DynamicVar 类创建的 EnvelopeVar 类, 可以执行更有趣的插值

```
struct SoundEnvelope
{
public:
    SoundEnvelope()
    { clear(); }

    void clear()
    {
        m_fInitialTime = 0.0f;
        m_fAttackTime   = 0.0f;
        m_fAttackLevel  = 0.0f;
        m_fDecayTime    = 0.0f;
        m_fDecayLevel   = 0.0f;
        m_fSustainTime  = 0.0f;
        m_fSustainLevel = 0.0f;
        m_fReleaseTime  = 0.0f;
        m_fReleaseLevel = 0.0f;
    }

    float m_fInitialTime;
    float m_fAttackTime;
    float m_fAttackLevel;
    float m_fDecayTime;
    float m_fDecayLevel;
    float m_fSustainTime;
    float m_fSustainLevel;
    float m_fReleaseTime;
    float m_fReleaseLevel;
};
```

```
class EnvelopeVar
{
public:

    enum Progression
    {
        ENV_FLOOR,
        ENV_INITIAL,
        ENV_ATTACK,
        ENV_RELEASE,
        ENV_SUSTAIN,
        ENV_DECAY
    };

    EnvelopeVar()
    { clear(); }

    void clear()
    {
        m_Envelope.clear();
        m_Level = 0.0f;
        m_eState = ENV_FLOOR;
    }
};
```



```
    }

    void setVar(const SoundEnvelope& env)
    {
        m_Envelope = env;
        m_eState = ENV_INITIAL;
        m_Level.setVar(
            m_Level, m_Envelope.m_fInitialTime);
    }

    operator float()
    { return m_Level; }

    void update(float fDeltaTime)
    {
        m_Level.update(fDeltaTime);
        if(m_Level.hasReachedTarget())
        {
            switch(m_eState)
            {
            case ENV_FLOOR:
                break;
            case ENV_INITIAL:
                m_eState = ENV_ATTACK;
                m_Level.setVar(
                    m_Envelope.m_fAttackLevel,
                    m_Envelope.m_fAttackTime);
                break;
            case ENV_ATTACK:
                m_eState = ENV_DECAY;
                m_Level.setVar(
                    m_Envelope.m_fDecayLevel,
                    m_Envelope.m_fDecayTime);
                break;
            case ENV_DECAY:
                m_eState = ENV_SUSTAIN;
                m_Level.setVar(
                    m_Envelope.m_fSustainLevel,
                    m_Envelope.m_fSustainTime);
                break;
            case ENV_SUSTAIN:
                m_eState = ENV_RELEASE;
                m_Level.setVar(
                    m_Envelope.m_fReleaseLevel,
                    m_Envelope.m_fReleaseTime);
                break;
            case ENV_RELEASE:
                m_eState = ENV_FLOOR;
                break;
            };
        }
    }
}
```

```
private:
    SoundEnvelope  m_Envelope;
    DynamicVar     m_Level;
    Progression    m_eState;
};
```



EnvelopeVar::update()函数通过枚举类型 Progression 转换自身的状态,并根据存储在 SoundEnvelope 结构体中的数据来从一个插值阶段转移到下一个插值阶段。然后就简单了,可以使用独立的对象,比如音量、音高或者音调,来表示你所希望其值随着时间流逝自动插值的变量,从而全面控制这个声音的完整包络线。为了更好地理解这个类的工作原理,可以运行随书光盘中的 SoundEnvelope 演示程序,听一听效果。

7.3.4 其他改进

根据你所采用的特定游戏引擎和环境的不同,你可能会在代码中以不同的方法使用这些类。比方说,你正在开发一款视频游戏,并且游戏引擎的模拟更新(非渲染)频率固定为 60Hz,你就可以考虑对 DynamicVar 类中的大部分做一些优化,通过预计算一些变量,使其只需进行简单的整数数学运算。例程中的代码比较适合 PC 游戏引擎,因为一般来说 PC 游戏的更新频率是变化的。

对于一些应用程序,可能想要让程序触发 EnvelopeVar 类中包络线的各种状态的变化,比如遇到播放动态序列时不能预知它的长度的情况。在这种情况下,只要简单增加一个方法阻止包络线不能超过某个特定状态(典型为保持)。另外应该意识到,本文中的代码只是为了阐述清楚概念。假如你打算在一个音乐音序器中对每个播放的声音都进行包络线控制,或假如你计划同时使用大量这些对象,显然这些代码还有很多优化余地。

7.3.5 结论

创建一个可以自行动态插值的自定义数据类型,其中的基本概念很简单,实现起来也很简单。然而这个概念在音频编程中的实际应用并不这么明显。通过创建编程工具和工具包,我们能够很容易地通过程序产生逼真以及自然流畅的音频输出,从而为音效设计师创造条件,使其得以将目前硬件能力发挥到极致。

7.3.6 参考文献

[Boer02] Boer, James, *Game Audio Programming*, Charles River Media, Inc., 2002.

[Meyers96] Meyers, Scott, *More Effective C++*, Addison-Wesley Longman, Inc., 1996.

[Meyers98] Meyers, Scott, *Effective C++, Second Edition*, Addison Wesley Longman, Inc., 1998.

7.4 创建一个音频脚本系统

作者: Borut Pfeifer, Radical Entertainment

E-mail: borut_p@yahoo.com

译者: 万太平

校对: 沙鹰

最近几年, 音频效果在游戏中的表现越来越突出。在此前相当长的一段时间中, 都没有支持高品质音效的技术。即使是在技术改进后, 音频仍在一段时间内被忽略, 没有将音频作为一种能令玩家沉浸于游戏中的工具来使用。如今已有多种工具可以帮助你游戏中创建出惊人的音效体验。但是就创建某种效果必需何种功能做出正确判断, 仍有相当难度, 尤其是对于新手而言。幸运的是, 存在一个相对简单的功能集, 可以通过脚本系统 (scripting system) 提供给作曲家和音效设计师们使用, 从而为你的游戏创建出有趣的音频效果。

本文描述了一个脚本系统, 通过控制用于回放的音频元素以及随机化环境效果, 实现使玩家沉浸于游戏的目的。该脚本系统通过使用一个基于XML的音频字典 (audio dictionary), 将音频标记ID与影响每次声音回放的控制参数进行绑定。其中音频一共分为三种基本类型: 音效 (effect)、音乐 (music) 和环境音效 (ambient audio)。三类音频各有各的, 为了有效地使用此类音频就必须要考虑的专属问题。该系统也允许在一组音频中进行随机播放。须知, 玩家往往打开游戏一玩就是好几个小时, 即便是最高质量的音效, 若是频繁地重复播放同一段, 也只会惹人生厌。

在游戏中使用音效通常有以下几个目的。首先是把正在游戏世界中发生的简单信息传达给玩家, 比如他们正在使用什么道具, 正处于什么类型的地区, 以及下一步在他们身上将发生什么事情。其次, 对话 (dialog) 一般主要用来表达游戏情节, 但对话在游戏中也能起到烘托气氛的作用。最后一个目的比较难达到, 那就是按照游戏的设计来影响玩家的情绪。游戏希望玩家在特定时刻感受到兴奋、平静、活泼或忧郁等等。

有几种技术可以用于达成最后一个目的。使用音乐来掌控玩家的情绪是一种非常强大的手段。而环境音效则使玩家沉浸在游戏想表达的叙述性的环境中。在我们生活中的每时每刻都充斥着通常不会被人有意识地察觉到的细微声音, 但正是因为存在这些声音, 我们才能够不断地感受到当前自己所处的位置。

一个良好的音频脚本系统应该能让音效设计师表达出音效的信息 (包括故事情节), 并能通过一种简单明了的方式来调整玩家在游戏中的心情。

7.4.1 游戏中音频的类别

在游戏中有几个音频类别，每个都有自己的技术需求、设计问题和脚本需要。这个脚本系统通过一个包含音频文件及其相关参数的数据库来控制不同类别的音频的播放。

这个数据库由一个 XML 文件组成，其中每个标识符（tag）对应你希望在游戏中播放的一段音频文件。一个标识可能是简单地指定一个 .wav 文件，也可能是指定一组文件用以循环播放，且组中每个都有一定的被选中概率。每个标识有一些参数用于控制和随机播放音频文件，为玩家创造更加多样化和独特的音效感受。

无论属于什么类别，每段音频都有一些控制或脚本方面的基本需要。例如，为了解决音频缓冲器数量有限的问题，每段音频文件需要有优先级。这样我们就可以决定，当请求新的音效而所有缓冲器都已满了的时候应当中断哪一段音效。若请求的声音优先级最低，它就不会替换任何当前播放的声音。其他的基本参数还包括音量调整范围（以随机地产生稍轻或稍响一些的声音）和以随机时间间隔循环播放的能力。

1. 音效

音效对应游戏中的变化，它们在游戏中通常只持续很短的时间。例如，这些动态的音效可以是枪声、呼噜和脚步声。因为这些效果由游戏中当前正在发生的事情决定，所以它们的响应速度必须很快，以便不论有无其他正在进行的处理，玩家接受到的视觉和音效方面的反馈都能够完美地保持同步。

因为大部分音效文件都短小精悍而且频繁地被播放，所以可以考虑将它们从磁盘中一次读入内存，然后通过音频引擎控制，在需要的时候播放适当的音效。（假如游戏中有些比较长而不重复的声音，你就可以考虑对磁盘上的一些声音文件采取流式（stream）访问的方法，而仅仅将最常用的声音文件存放在内存中。）当播放的时候，内存中每个声音文件都装载到一个单独的声音缓冲器中，这样声音库能够在它上面做一下处理来对声音进行空间定位。这是因为，即使是两个相同的声音，但因其离开玩家的距离不同，听上去也应该有细微的差别。

当一个音效文件被快速地连续播放多次，几个复本同时播放出来，可能产生一种令人不愉快的音频失真，称为镶边效果（flanging）。这种声音失真的产生是因为相同的声音在它的相位外播放造成的。在游戏中这是很常见的情景，例如数个敌人正在使用相同的武器开火。为了防止镶边效果的产生，你可以在某个音效文件重复播放达到指定次数的时候，选择一个二级的声音文件播放。这个音效可以是级联的（cascade），也就是说新的音效又有自己的二级音效。例如，由于多次播放打碎一小块玻璃的声音不能充分模拟打碎一大块玻璃的声音，所以当请求播放多个打碎小块玻璃的音效时，可以用打碎一块较大的玻璃的音效替代它们。同理，当请求播放打碎多个大块玻璃的音效时，可以用打碎一块更大的玻璃得声音来替代。这也帮助减少了被该组相关音效占用的硬件声音缓冲器的数量。

2. 音乐

交互式音乐在游戏行业中正受到越来越多的关注。像 DirectMusic 这样的工具提供完全分层的音轨（soundtrack）的功能，甚至还可以动态调整单件乐器合成到最终配乐中的量。其

复杂度之高若此，看起来可能很是吓人。但以交互式音乐著称的游戏 *Halo* [O'Donnell02] 使用了一个相当直观的系统来控制游戏中播放的配乐。

每部乐曲 (composition) 都由三个子块组成：一段音乐在作品开始的时候播放，一段持续一定时间的音乐用来循环播放，最后一段音乐当循环结束时播放。每个单独的音乐段都可以利用控制和随机化的标识符，这样就可以在作品播放的时候实现更加广泛的音乐感受。脚本系统只需要指定播放的音乐作品，因此提供一个非常简单的接口就行了。另外，每部作品需要指定在循环音乐和退出音乐之间的转换是如何处理的。系统可以在指定时间段内在两段音乐之间实现淡入淡出效果 (cross fade)，也可以立即切换。在 [O'Donnell02] 的系统中使用了一系列可以单独在脚本系统中使用的额外的循环和退出标识符，虽然其中的一些功能也能够使用其他某个相近的作品标识符来重现。

创建成功的交互式配乐的关键之一就在于，在并不允许玩家直接控制当前播放的音乐的前提下确保音乐符合玩家行动时的情绪。赋予玩家太多的控制权反而会使得他们感到不连续，因为玩家是以你创建交互配乐的规则而开始游戏的。举个例子，若是玩家一进入某个令人毛骨悚然的走廊，音乐就变得非常的戏剧性，而当玩家掉转头来音乐就立刻变得不是那么戏剧性，这样一来玩家沉浸就不协调。作曲的人需要能够帮助定义玩家整个的情绪，而不是基于玩家的一时兴奋而每时每刻调整。

另一个需要考虑的关键问题是，大部分的游戏长度都在 20 小时以上，可是仅有很少的几款游戏含有的独特音轨总长度超过一个小时。如果在整个游戏过程不停地播放音乐，就可能导致重复度非常高的音乐体验。因此，假如游戏采用音乐的目的是提升玩家的情绪，就不能滥用音乐，需有节制地使用，以保持音乐的情绪冲击力。

3. 环境音效

环境音效 (ambient effect) 也是通过空间定位的，在这点上它类似于其他更加普通的音效。环境音效主要为游戏创建音响空间。正是那些不起眼的音效，例如沼泽地里的蟋蟀鸣叫以及城市街道上嘈杂的人声等，帮助玩家沉浸在游戏里面。环境声不像其他音效那样需要对变化做出响应，而且由于单个环境声片断可能较长，它们最适合在运行时候采用流式处理。

为了在游戏中创建一个环境音响空间，脚本的标识应当支持简单的随机播放和分组播放。例如，一个有关丛林的环境声的脚本标记可能包括蟋蟀鸣叫声、风的呼啸声以及其他多种动物发出的声音。这些元素可以各自按照不同的随机频率循环播放，也可以随机地移动其在三维空间中的位置。这样就可以确保环境声不会以相同的次序重复播放，否则玩家会很容易察觉到重复。

4. 对话

游戏中的对话可以归为两个主要的类别，过场动画中的对话和游戏中角色之间的响应式对话。过场动画对话的控制很简单明了。对话不必像其他音效那样总是存放在内存中，大部分的对话也不必使用 3D 定位音频。过场动画对话总是从中心的声道出来，这是因为假如对话声被处理成相对玩家位置而定的 3D 立体声，那么当玩家在距离稍远的位置触发过场动画的时候，就有可能听不清其中的对话。不过你可能仍然想根据过场动画发生的位置来处理环境音效，例如混响 (reverb) 和回音 (echo)。过场动画对话的主要功能就是把故事情节说给

玩家听。

游戏里面的对话也帮助展开故事情节，但这一般是通过设定或者游戏氛围来达到的。我们既可以利用对话中的辱骂和反诘表现角色的愤怒，也可以在角色觉得应该逃跑的时候通过对话表现出恐惧。这样即使对话中没有太多具体故事的细节，但因为有了更广泛的沟通和表达的途径，仍然可以使敌人和队友看起来栩栩如生。对话尤其有助于传达一些由游戏中 AI 做出的决定，从而玩家能够理解他们正身处其中的游戏世界，并相应地制定游戏策略。这种类型的对话是和游戏类别有关的。游戏 *Halo* 使用一个系统，其中含有有限的几个录制好的海军陆战队员的声音。每个陆战队员对于不同的状况有不同的口头禅；假如一个陆战队员倒下了，就尽量避免在游戏中重复选中他的声音。你的游戏通常对游戏中的对话会有独特的需求，比如不同类别的回答，所以应该基于你的游戏设计来进行游戏对话的需求分析。

7.4.2 工具



ON THE CD

随书光盘中的例子是使用微软公司 DirectX 中的 DirectSound 来实现的。DirectSound 提供了包括 DirectMusic 在内的广泛功能，但我们将把它用在基本的底层声音播放和 3D 音效方面。假如 DirectSound 还不能满足你的全部需要，你可以考虑使用一些其他类似的音频函数库，其中有免费软件也有需要付费的商业软件，它们可以在各自不同的平台上实现相似的功能。但，你必须自行移植那些代码。

我们用 XML 来定义音频标识的数据库。定义和加载必要的音频参数是非常简单的。另有一篇文章讨论将 XML 用于游戏开发中其他方面的问题 [Seegert02]，而且我们还可以在网络上找到很多相关资源，所以本文并不打算详细描述 XML 的用法。

我们使用 Ogg Vorbis SDK 来进行音乐回放 ([Moffit02]、[OggVorbis])。也很容易将对其他格式的支持添加到本例的结构当中（也就是继承一个新的 C++ 类，Ogg Vorbis 功能正是一个范例）。Ogg 编码在音质方面可以与 MPEG Layer 3 (mp3) 编码相媲美，而且实现解码器 (decoder) 无需支付任何许可证费用。因此 Ogg Vorbis 在音乐回放方面，是一个很有吸引力的解决方案，特别是对于那些囊中羞涩的独立开发者而言。

7.4.3 基于 XML 的音频标记库

标记数据库文件包括一系列可以由游戏引用的音频标记 (audio tag)。一个音频标记就对应于音频设计师希望在任一点播放的一个音频逻辑段（实际上可以由数个不同的音频文件组成）。多个标识符以如下的方式构成标记数据库文件：

```
<?xml version="1.0"?>
<AudioTagDatabase>
...
</AudioTagDatabase>
```

标记分为 6 大类：音效 (EFFECT)、环境声 (AMBIENT)、音乐 (MUSIC)、乐曲 (COMPOSITION)、组 (GROUP) 和随机 (RANDOM)。下面给出标识符的格式范例，从中

可见有一些属性是这 6 类标识共有的（表 7.4.1）。

```
<TAG_TYPE_NAME ID="FOO" PRIORITY="0"
  VOLUME_ADJUST="5.0" VOLUME_ADJUST_RANGE="3.0"
  LOOP_DELAY="10.0" LOOP_DELAY_RANGE="5.0"/>
```

表 7.4.1 各类标识共同的 XML 属性

ID	字符串，游戏代码和脚本通过 ID 要求音频引擎播放所希望的声音。
PRIORITY	整数，反映该音频标识的重要程度，即优先级（在所有的缓冲器都被占用的时候，它被其他声音打断的可能性）。优先级越高，能够打断它的声音就越少。默认优先级为零。
VOLUME_ADJUST	在游戏默认音量上调整声音播放的分贝数。这个调整限制在最大和最小音量之间。正值增大音量，负值则减小音量。
VOLUME_ADJUST_RANGE	随机增加或减少音量调整的音量分贝范围。例如 VOLUME_ADJUST 是 3.0，而 VOLUME_ADJUST_RANGE 是 1.0，则每次播放音效的时候，音量实际的调整在 2 分贝和 4 分贝之间。
LOOP_DELAY	该声音重复播放的时间间隔（秒）。假如没有指定本参数，或指定为负数，则该声音不会被重复。
LOOP_DELAY_RANGE	随机调整重复播放延迟时间的上下范围。
LOOP_TIMES	循环播放该声音的次数。假如这个值为 0 且指定了一个 LOOP_DELAY，就意味着该音频文件将不停地循环播放。

1. 音效

音效标记有一些额外的参数，如表 7.4.2 所示。

表 7.4.2 音效标记的 XML 属性列表

FILE	该音效需播放的.wav 文件名
MINDIST	假如玩家和声音之间的距离比 MINDIST 近，将听不见这个声音
MAXDIST	假如玩家和声音之间的距离比 MAXDIST 远，也听不到这个声音
CASCADENUM	在激活级联音频标记之前，这个声音需要被同时请求的次数
CASCADETAG	当音频管理器（audio manager）检测到有一定数量的该标记正在同时播放，就会播放这个级联音频标记

2. 环境声

环境声标记也属于音效标记，所以可以适用任何音效标记的参数。另外环境音效标记还具有表 7.4.3 中列出的这些参数。

表 7.4.3 环境音效标记的 XML 属性列表

X	声音在世界中的 x 坐标
Y	声音在世界中的 y 坐标
Z	声音在世界中的 z 坐标
XRANGE	在每次播放的时候随机调整声音位置的 x 坐标的范围。例如一个环境音效标记的 x 位置为 5.0 而 XRANGE 为 10.0，则声音位置的 x 坐标在-5.0~15.0 之间变化。单位采用游戏中的基本单位。
YRANGE	在每次播放的时候随机调整声音位置的 y 坐标的范围
ZRANGE	在每次播放的时候随机调整声音位置的 z 坐标的范围

3. 音乐

音乐的标记只有一个参数，就是播放文件的名字（见表 7.4.4）。

表 7.4.4 音乐标记的 XML 属性列表

FILE	以流式载入该段音乐的.ogg 文件的名称
------	----------------------

4. 乐曲

乐曲标记描述一个由三段组成的循环音乐结构：一段音频流作为开头，一段循环音频，一段音频流作为收尾（参考表 7.4.5）。

表 7.4.5 乐曲标记的 XML 属性列表

IN	请求乐曲文件时，首先开始播放的音频标记
LOOP	IN 标记播放完成后，开始循环播放 LOOP 标记，根据乐曲的长度决定循环持续的时间
OUT	LOOP 标记播放完成后，开始播放 OUT 标记
CROSS_FADE_TO_OUT_TIME	这个就是从 LOOP 标记结束播放（开始淡出）到 OUT 标记开始播放（开始淡入）的时间。假如该参数为 0 或者干脆没有指定，就是说没有任何淡入淡出的效果，在 LOOP 标记播放结束后立刻开始播放 OUT 标记

5. 组

这个标记本身没有任何属性，但可能带任意数目的子标记，格式如下：

```
<ITEM TAG="TAGNAME" DELAY="delay time" DELAY_RANGE="delay range"/>
```

当播放 group 标记时，每一个通过子项指定的标识都同时开始播放。假如其中某个子标记指定了 DELAY 参数，系统就会等 group 中的其他子标记开始播放之后，延迟一定时间才开始播放这个子标记。DELAY_RANGE 参数按照一定的随机量来调整延迟的范围。这里的时间单位都是秒。

6. 随机

和 GROUP 标记类似，这个参数没有任何属性，不过也可能带任意数目的子标记，格式如下：

```
<ITEM TAG="TAGNAME" PROB="percentage 0-100">
```

虽然可以有任意多个子标记，所有子标记的 PROB 属性值之和不能超过 100。这个值代表了当 RANDOM 标记被使用时某一个子标记被选中播放的可能性；因此，和大于 100 是说不通的。

7.4.4 脚本系统组件



随书光盘中几个例子的实现用到了一些基本的组件。例子的代码在 Microsoft Visual C++ 6.0 里编译通过。

1. AudioManager 类

这是一个单子（singleton）类，是游戏中封装音频引擎的主要接口。它负责分配硬件声音缓冲器，定时地将其更新为当前播放的声音。但是由于声音缓冲器的容量是有限的，所以只能分配固定个数个，并在继续播放时用声音数据填充它们（假如声音长于缓冲器，则对缓冲器做循环）。表 7.4.6 列出了一些重要的函数。

表 7.4.6 AudioManager 类的主要接口函数	
LoadAudioTags	接受一个音频配置 XML 文件名作为参数。
Play	接受的参数有：音频标识的名字、一个 WorldObject 的指针（用于声音的音场化，可以是 NULL）、播放持续的时间（单位是毫秒，0 表示声音文件的正常持续时间）、延迟（等待播放声音的时间毫秒）和一个 IAudioListener（它会收到声音播放结束的消息，也可以是 NULL）对象指针。
Update	接受一个自从上次更新后经过的时间毫秒数作为参数。腾出声音缓冲器的一部分，这样播放中的声音可以载入各自的下一部分。
SetOverallVolume	调整参数，覆盖默认值。 SetDistanceFactor SetDopplerFactor SetRolloffFactor
SetListenerCamera	接受一个 Camera 对象的指针作为参数。这个函数允许 AudioManager 为一些需音场化的音效更新三维监听位置。

2. Audio 类

Audio 类代表特定类型的音频，并且懂得怎样将数据装载到由 AudioManager 管理的缓冲器当中。它通过虚函数 FillBuffer()来工作。该函数将音频数据转换成 AudioManager 可以使用的格式（在这个实现中，DirectSound 使用 PCM 数据）。它也让 AudioManager 知道是否有更多的音频有待播放。

Audio 类有几个子类。Sound 类通过 FillBuffer()接口来装载.wav 文件。而它的子类 Sound3D，则指定游戏世界中一个有具体位置的对象以将声音关联上去。类 MusicOggVorbis 实现了一个流式使用.ogg 文件的音频对象。它继承自 Audio 的子类 Music 类。Music 类是所有音乐文件类型的基类。

3. IAudioListener 接口

任何类假如希望在一个特定的音频段播放完毕的时候接获通知，都需要继承这个抽象类。该类仅有一条虚函数，如表 7.4.7 所示：

表 7.4.7 IAudioListener 接口类中的纯虚函数	
AudioFinished	当期望的音频文件播放完毕，AudioManager 就回调这个函数。它带一个参数，就是那个播放完毕的 Audio 对象的指针（因为可能该 IAudioListener 类正在等待多个 Audio 对象播放完毕）。

4. AudioTag 类

AudioTag 是任何在 XML 库中定义的音频标记的基类。在这个类中的每个对象都有三个

属性：一个优先级，即当要播放一段新的音频而没有足够的声音缓冲器可用时，该段音频被中断取出的可能性；还有两个参数用来定义应该怎样调整所有以该音频标记创建的音频的音量。它有三个能够被子类重载的函数，列在表 7.4.8 中。

表 7.4.8 AudioTag 类的主要接口函数

LoadTag	从 XML 文件元素中载入标记（Tag）数据。
CreateAudio	这个函数创建一个 Audio 合适子类的对象（例如 AudioEffectTag 创建一个 Sound3D 对象）。这个函数分配的内存当 Audio 对象结束播放时由 AudioManage 负责清除。每个子类都必须实现这个函数。
AudioFinished	AudioTag 类是 IAudioListener 接口的子类。本函数的默认行为是：若声音需要循环，就在播放结束之时创建声音的另一个实例。子类可以重载这个行为。

AudioTag 的每个子类(AudioEffectTag、AudioAmbientTag、AudioMusicTag、AudioCompositionTag、AudioGroupTag 和 AudioRandomTag)都有自己的数据成员对应音频库 XML 文件(audiodb.xml)中每个标记的参数。

5. WaveFile 类

WaveFile 类是 DirectSound SDK 例程中某一个同名类的简单版本。类 WaveFileFactory 处理 WaveFile 对象的内存管理。在本例中，它为所有请求加载的.wav 文件分配内存，但这个内存管理的工厂模式(factory)可以更加复杂。比方说，你可能希望根据播放频率和内存需求，仅仅将一定数量的文件存放在内存中。

6. OggVorbisFile 类

OggVorbisFile 类是从有良好文档的 Ogg Vorbis 范例([OggVorbis], [Moffit02])中得来的。它简单地封装了 Ogg Vorbis SDK（具体地说，就是主要负责读取.ogg 文件的 vorbisfile 库）。

7. Camera 类

这个类描述一个游戏用来跟踪玩家的视野的对象。它包含摄像机(Camera)的位置、一个描述摄像机朝向的向量、另一个描述摄像机在垂直方向上的倾角的向量。AudioManager 为了实现 3D 音频定位，使用这个信息来更新 DirectSound 的 3D 听众。

8. WorldObject 类

这个类描述了一个在 3D 游戏世界中有具体位置的游戏对象。空间定位的声音通过相关联的 WorldObject 对象来更新其 3D 位置和速度。函数 GetPosition()、SetPosition()、GetVelocity() 和 SetVelocity()用来设定这个数据。

9. 脚本控制

任何音频标记都可以在脚本中被调用，也可以与对象、动画、粒子系统、角色、位置等建立关联。惟一需要的接口是 AudioManager 类中的 Play()方法。

10. 范例应用程序



随书光盘上附带的例程有着非常简单的 GUI, 允许你控制你可能在游戏中用到的音频参数。你可以在库中找出任意一个音频标记来播放, 调整听众的位置和总音量等。感受一下不同的环境、音乐和音效标记吧, 看看不同的标记类型和参数是怎样相互作用而形成一个令人陶醉的音场的。

7.4.5 进一步的工作

这个 XML 脚本系统有很强的扩展性。系统中没有针对游戏中对话提出要求, 比如将镜头的对话改成不进行空间化的, 这是因为游戏中对话的要求都是基于特定的游戏设计的。另外这个脚本系统在增加不同类型的标记和参数方面也非常方便: 比如一个仅仅用于 2D 的音效参数, 或者包含一个特殊的对话标记——由一个由分属数种回应类型 (例如被击中、垂死等) 的对话行组成的矩阵作为参数。

有些类型的游戏, 可能要求音效动态地对游戏中的环境做出调整 (例如汽车引擎在加速过程中会不断改变音高)。这可以很容易地实现为另一个空间化的音频标记。另外像回声和混响这样的 3D 效果, 还有那些控制声音受影响的随机性因素, 也可以作为 EFFECT 标记的参数。

7.4.6 总结

这组简单的脚本化的音频属性, 使你能够快速开发更具交互性的游戏音响效果。现有的技术 (例如 XML) 和编辑这些属性的工具配合使用, 在进行数据定义时效果很好。

7.4.7 参考文献

[DirectX] DirectX 9.0 SDK DirectSound examples, available online at www.msdn.com/directx.

[Moffitt02] Moffitt, Jack, "Audio Compression with Ogg Vorbis," *Game Programming Gems 3*, Charles River Media, 2002.

[O'Donnell02] O'Donnell, Marty, "Producing Audio for Halo," available online at www.gamasutra.com/resource_guide/20020520/odonnell_01.htm, May 20, 2002.

[OggVorbis] Ogg Vorbis SDK, available online at www.vorbis.com.

[Seegert02] Seegert, Greg, "Real-Time Input and UI in 3D Games," *Game Programming Gems 3*, Charles River Media, 2002.



7.5 使用 EAX 和 ZoomFX API 的环境音效解决方案

作者: Scott Velasquez, Gearbox Software

E-mail: scottv@gearboxsoftware.com

译者: 万太平

校对: 肖丹

假如你正在读这篇文章, 你可能只是音频程序方面的新手, 但分配到令人畏缩的任务, 要增加环境音效 (environmental audio) 到你当前的项目中。或者虽然你是音频程序方面的老手, 但仅仅擅长 2D 音频。无论哪种情况, 本文对你都很适用。

这篇文章主要介绍在游戏里面实现环境音效方面的一些想法和建议。代码实例使用 DirectSound API 和 C++ 语言编写, 运行在 Windows 平台上编译。作者假定读者对于 DirectSound 和 DirectSound3D 有些经验, 因此在本文中就不对 DirectSound 作太多的介绍, 它超出这篇文章的范围。假如读者对于两者都有经验的话, 例程中的代码可以使用其他语言或者对于不同的音频库重写, 尽管 ZoomFX 扩展只对于 DirectSound 有用。

7.5.1 什么是环境音效

在 3D 空间中把声音正确定位好之后, 要创建一个使玩家流连忘返的世界, 你能够做的最重要的事情就是创建一个带环境属性的环境音效, 来影响你游戏中的音效。James Boer 做了一个非常好的比较: 环境音效对于声音的作用就如同光影在图形渲染中的作用 [Boer03]。

当以声音从音源传递到一听众, 音效会因为传播通过的通道环境影响而改变。声音能量可能被通道中的物体反射或者吸收。

环境音效就是应用软件来模拟这些效果。通过把各种环境音效, 比如回音、混响和频率滤波器, 应用到最初音频样本上来做这些调整。回声和混响 (也分别称为 “早” 和 “迟” 的反射) 提供给玩家的信息包括音源距离多远和它是从什么类型的环境中发出的。直接到达听众 (没有任何阻塞) 的声音被认为是 “直接声音 (direct path sound)” (如图 7.5.1)。

频率滤波器通过移去样本高频成分的一部分, 对游戏内物体阻塞声音的现象生成消声效果。滤掉的总量由阻塞声音对象的成分决定, 其中一个重要的因素就是材质属性 (参见图 7.5.2)。

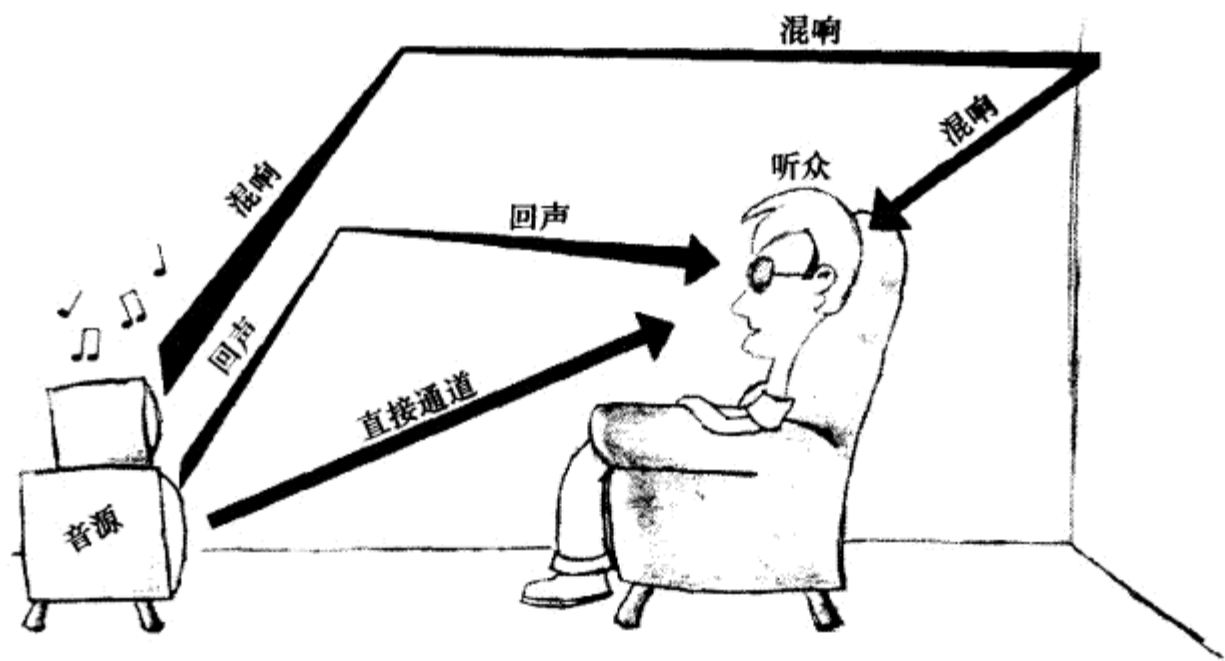


图 7.5.1 直接通道、回声和混响

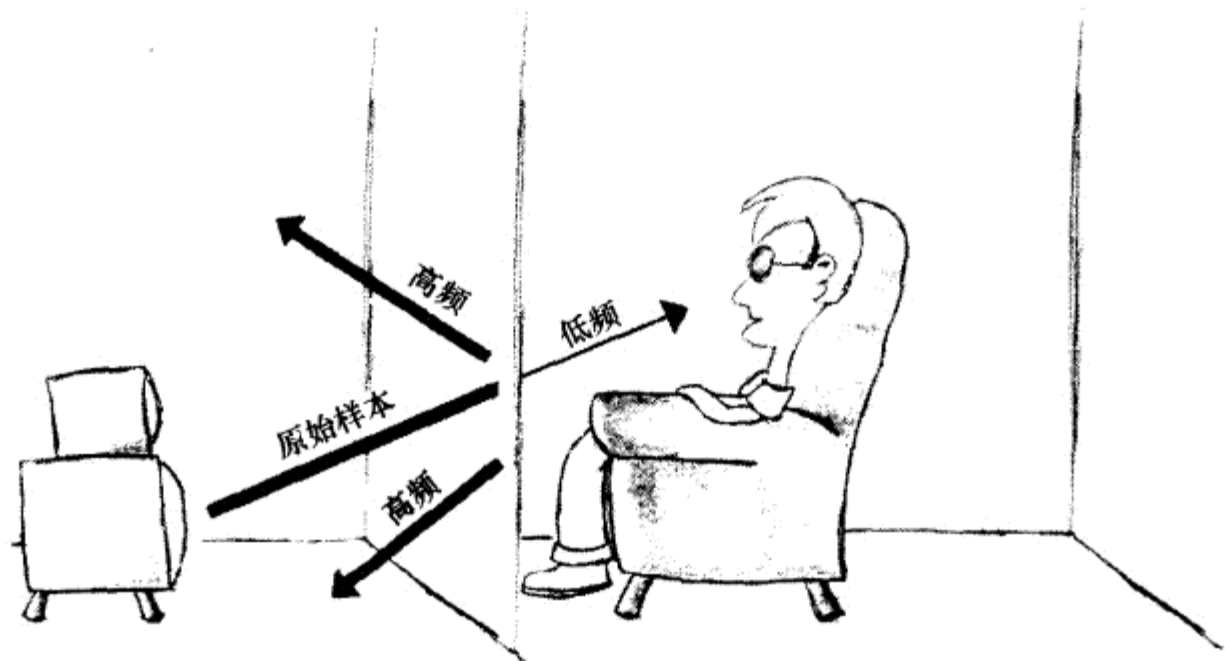


图 7.5.2 封闭/遮断示意图

滤波也应用到如角色对话这种具方向性信息的声音当中。说话者的声音是向前面方向扩散的，所以位于你身边和后面的人能听到的高频部分要少很多。这些效果合起来工作会生成大家知道的混响模型（reverberation model），在以后的章节介绍 Creative Labs 公司的 EAX（Environmental Audio Extensions）的时候，我们会详细讨论。

时至今日，大部分的声卡都有在硬件里面混合这些效果的能力。遗憾的是，一些整合芯片（比如在很多新的 Dell PC 上使用的 SoundMAX）不能在硬件上执行基本的 3D 空间定位的效果，更不必说计算完全的环境音效。因此，当考虑音频的特性时，时刻应该牢记，环境音效应该考虑为高级效果，假如用户的硬件不支持的话，不应该打破你的游戏性。

7.5.2 音频引擎的系统要求

对于环境音效系统来说，软件需求最困难的部分是有效存储和快速获取空间数据。基于高细节度的几何来创建可信的和动态的听觉环境效果，对于处理或者存储都不便宜。现在大

部分程序员都能有效地使用 BSP (Binary Space Partitioning) 结构用于存储和取回图形数据,而在 BSP 出现之前图形程序员所处的困境正是当今音频程序员所经历的另一困境。

在音频引擎中实现这部分的困难还需依赖你在空间检索方面的经验(空间检索技术已经超出本文的讨论范畴),在你的游戏引擎中使用了什么空间排序技术(大部分很可能用于渲染或者 AI),在影响你的美工内容创建途径方面你有多少控制能力。

1. 存储和取回数据

有很多方法用于存储你的环境数据。基本上说来,任何 3D 图形存储系统都可以作为存储音频空间数据的选择,包括 BSP 树、四叉树(二维空间的树,每个父节点有 4 个子节点),或者八叉树(三维空间树,每个父节点有 8 个子节点)。以及 Creative Labs 设计的应用程序 EAGLE (Environmental Audio Graphical Librarian Editor)。EAGLE 使用简单化的几何体来存储环境数据(可以通过一个外部 DLL 来取回到你的游戏中)。

对于每帧,基本上需要做三件事情来存储和访问:

- 音效的环境数据(玩家当前的环境和最后一帧用于环境渐变的环境数据);
- 潜在可以听到的声音集(玩家潜在可能听到的声音);
- 用于计算封闭和遮断时相关对象的材质数据。

2. 存储 EAX 环境数据

需要一种方法来存储环境数据,这样当描述听众和音源环境的时候你可以传输给 EAX。在上一个项目中,我们通过创建包含环境属性的入口(portal)技术来处理这个问题。这个对于已经存在渲染系统的游戏可以工作得很好。你应该采用的方法则是依靠你的游戏。比如赛车游戏能够通过使用四叉树或者格子状的结构用于赛车的轨迹,使用一个 3D 结构用于表示隧道和其他特殊的结构,侥幸成功。当做选择的时候,显然你希望使用最快的方法而且只要最少的内存,但你也需要考虑它会怎样对需要输入数据的那些人,比如关卡师或者音效设计师等,产生多大的冲击。

7.5.3 潜在可听集 (PAS, Potentially Audible Set)

你可能听说过潜在可见集 PVS,它用于描述渲染的时候可能可见的对象。PAS 的思想和 PVS 一样,但它用于音源。Quake 游戏引擎称它为 PHS (Potentially Hearable Set)。这一定义所指的是在每个指定位置可以达到听众的声音集。

这个功能可能已经是你的声音引擎里面的一部分了。假如没有,就需要注意没有任何魔法般的公式或业界标准方法可以计算 PAS。PAS 计算本身就有自己的话题。我们已经涉及到的引擎使用图形渲染器(render)的 PVS 数据。有相当多的人在这个方面做了很多的研究,同时 3D 音频现在在视频游戏中变得重要起来。减少花费较多的运行时计算和创建一个更加真实的体验是我们努力的重要目标。

1. 实现一个真正的 PAS

当构建它的一系列潜在可听到的声音集时,少数几个游戏在创建 PAS 结构时考虑声波的反弹

和其他声音相关属性。实现一个 PAS 结构需要做一点工作和并需要额外的内存空间，因此在你的目标平台上可能会也可能不会考虑。然而，它能够更加精确地表示潜在可听到的声音集，因为它考虑声音相关的属性，比如声波反弹属性和不同材质吸收不同的能量。一个真正的 PAS 同样创建一个简单化版本的几何形状，减少存储的内存空间和增加用于检测遮断的射线跟踪的计算。

2. 使用图形渲染器的 PVS 结构

一些人可能已经倾向选择一个虽然没有那么真实但需要做的工作较少的方法（因为时间或者内存限制），可以通过使用图形渲染引擎的 PVS。例如 Quake 2 PAS 结合当前的 PVS 以及从当前 PVS 的入口点看过去按顺序的第二个 PVS。这样使得数据集包含比标准的 PVS 更多的区域。这个结构不是第一次被其他系统使用，因为 AI 和物理模拟也都需要它。虽然使用 PVS 来检测可能听到的声音集效果不错，但随着 3D 音频技术的发展，我们最终也会像其他技术一样放弃使用它。使用 PVS 惟一的不利之处就是，你被迫要对所有层级的表面形状都做光线跟踪。假如你有一个简单的集合，这个计算将会更快。

3. 计算声音的封闭（obstruction）

你选择什么样的 PAS 实现方法决定你怎样计算封闭效果。假如你决定使用 Creative Labs 的 EAGLE 程序，则它已经为你做好了这个工作，你只需要做一个调用把听众位置和音源的位置传递到 EAX Manager 就行了。

另外，无论你选择哪种 PAS 实现方法，处理过程本质上都是相同的。基本上，你将检查音源是否在 PAS 集中。假如不在 PAS 集中，音源被封闭。你可能选择容易的方法仅仅分配一个默认的封闭值，但我们不会满足于默认的那种效果。假如在计算封闭的时候考虑到材质的话，玩家会获得更加真实的效果。例如，砖块房子的封闭效果应该和清水墙的房子封闭效果不同。

为了使封闭声音的效果更加具有真实感，有好些事情可以做。你可以计算出在到达听众的通道上有些什么障碍物。这个很简单，只要你朝听众方向做一个射线跟踪和从听众到音源做一个射线跟踪就行了。然后，你计算出它们两者都碰到的对象预设材质（或者纹理）。当介绍 EAX 的时候会介绍材质预置。根据你的偏好，你可以结合两者或者仅仅优先选择材质比其他更加重要。另外可以做的事情是调整音源到听众之间的距离。选择任意一种方法，封闭效果都要比选择相同总量值的封闭效果更加令人信服。

假如音源在 PAS 集中，你选择必须做射线跟踪来决定音源在到达听众的时候是否和任何对象碰撞。假如有碰撞，声音被封闭。对能够传递声音的对象，在计算封闭效果时使用材质计算是有用的。计算封闭效果最好的方法（但不幸的是这很消耗时间）是测定可以通过封闭的角度。在大部分游戏中，这个代价未免太大，对听众发射一或两个设计跟踪的方法最常用于检查碰撞情况。

4. Creative Labs 公司的 EAGLE

在存储和取回空间数据部分我们提到 EAGLE。我们可以选择 EAGLE 允许内容创建者为游戏设定 EAX。使用 EAGLE 需要很少的程序支持，除非地图编辑器不能输出一个 EAGLE 支持的格式：3D Studio MAX (.ase)、LightWave (.lwo)，或者 DirectX Mesh (.x)。EAGLE 也包含一个 SDK 可以用于编写需要定制的输出器。这一选择所带来的惟一真正的不利方面是每个等级都需

要一个单独的处理过程,这依赖于分级几何结构。因此任何世界表面实质改变将需要打开 EAGLE 和调整环境设定。EAGLE 将创建一个 BSP 文件,在游戏中 EXA Manager 运用这一文件计算环境属性。

5. 关卡和音效设计工具

因为任何特征都需要内容支撑,所以你需要创建一个界面给音效和关卡设计师,方便他们利用这一接口来创建和安放将被 ZoomFX 使用的环境材质预设以及音量。这个界面应该尽可能简化以适应当前使用的流程。

如果由美工来做决定的话,他们应该会优先增加功能到世界创建工具中,而不是增加一个独立的新工具。假如你的游戏碰巧使用入口用于渲染,你可以把环境设定结合在一起。否则你将需要创建一些系统划分你的世界来表示环境。预设材质(material preset)很容易添加,因为你的游戏应该拥有一些方法把材质指定到面或者画笔上。

7.5.4 EAX 介绍

EAX 是 Creative Labs 公司开发出来的一组音频扩展集。这部分介绍 EAX 的接口,以及它是怎样通过 DirectSound 来访问的。

1. DirectSound 基础

在你开始重新编写有环境音效的音频引擎之前,必须首先理解 EAX 接口怎样和 DirectSound 一起工作。尽管 DirectSound 自从 DirectX 5 以后就没有多大改变(除了增加基于软件的 I3DL2),Direct3D 工程师因为 DirectSound 提供的一个特征嫉妒我们音频工程师,属性集(Property Set),Direct3D 中不提供。

DirectSound 开放基于 COM 接口的属性集,可以访问比如像 EAX 和 ZoomFX 音效这样不是 DirectSound API 的部分硬件特征。这个能力使 DirectSound 是可以扩展的,它通过允许硬件生产商创建包含新特性的属性集。每个属性集由硬件生产商创建的惟一的 GUID(Globally Unique Identifier)来指定。你将很快学会我们做的任何事情是怎样通过这些接口直接对于音频硬件访问。

2. EAX 基础

EAX 属性集提供可以调整的参数,当创建你的环境音效时,通过 EAX 混响引擎在内部创建声学方面的模型。EAX 1.0 和 EAX 2.0 主要不同之处不是混响引擎,而是通过 EAX 接口开放的属性。EAX 2.0 使得音频工程师和设计师有更多的旋钮可以调节。

在使用 EAX 工作的时候有两个基本的属性集:听众和音源属性集。在你的声音引擎中,对于每个 3D 声音缓冲器至多有一个听众属性集合和一个音源属性集。当设计你的系统时,确定记住了声卡最小的需求。假如你需要承受那种不支持任何 EAX 的硬件,你将想以这样的方法来对你的对象结构化,这样就不要存储那些从来用不到的 EAX 对象。

使用 EAX 属性时有几件事情应该做。类似 DS3D, EAX 提供延迟机制,允许你改变 EAX 属性和对执行的改变做延迟。强烈推荐设定延迟属性,因为单独地交付每个设定将使得多个系统

调用声卡驱动程序，并造成每个缓冲器会碰到多次不必要的工作。另外需要处理的事情是在设定属性时检查它的范围。假如你设定一个属性超出它的范围，EAX 调用失败，并且根据驱动的不同，或者限定（clamp）这个值或者干脆不设定它。

3. EAX Listener 属性集

类似 DirectSound 中 IDirectSoundListener 接口对象，EAX 听众属性集负责描述玩家所在游戏世界里面的周围环境。在 DirectSound，有一个主缓冲器，代表听众。当激活 3D 模型时，这个主缓冲器包含如位置、方向和速度等属性。EAX 建立在这些概念之上，另外增加帮助描述听众的环境的听众属性。这能取得极好的音效：比如提供描述听众所在房间的大小的能力，房间中反射和混响的音质，不同环境之间的渐变。听众（对于 DirectSound 和 EAX 两者都是）代表全局效果，可以应用到最终的混合中。EAX 的每个版本都有点差别，但它们都支持至少几个用于描述听众的环境属性。

一旦已经创建标准的 DirectSound 接口，你可以从 DS3D 次级缓冲器中的一个获得 EAX 听众属性。你可能觉得有点奇怪，因为 DirectSound 的古怪阻止主缓冲器的使用，你不得不使用次级缓冲器来取回听众属性集。为了取得听众属性集接口（IKsPropertySet），你需要在 3D 次级缓冲器中的一个上面调用 QueryInterface() 函数。

```
LPKSPROPERTYSET pEAXListener = NULL;
if ( FAILED(hr = p3DBuffer[0]->QueryInterface(
    IID_IKsPropertySet, (void**)&pEAXListener))
    || pListener == NULL )
{
    printf("QueryInterface failed to obtain EAXListener property set interface!");
}
```

4. 查询是否支持 Listener 属性集

在开始设定 EAX 听众属性前，你需要保证用户的声卡支持 EAX 听众属性集。

```
ULONG support = 0;

if( FAILED(pEAXListener->QuerySupport(
    DSPROPSETID_EAX_ListenerProperties,
    DSPROPERTY_EAXLISTENER_ALLPARAMETERS,
    &support)) )
{
    // 不支持 EAX 听众属性
}

if( (support &
    (KSPROPERTY_SUPPORT_GET|KSPROPERTY_SUPPORT_SET)) !=
    (KSPROPERTY_SUPPORT_GET|KSPROPERTY_SUPPORT_SET) )
{
    // 不支持 EAX 听众属性
}
```

5. 设定和操纵听众的环境

假如用户声卡支持 EAX 听众属性集, 现在你就能够设定一个属性。

```
EAXLISTENERPROPERTIES environment = EAX30_PRESET_AUDITORIUM;
if ( FAILED(pEAXListener->Set(
    DSPROPSETID_EAX_ListenerProperties,
    DSPROPERTY_EAXLISTENER_ALLPARAMETERS, NULL, 0,
    &environment, sizeof(EAXLISTENERPROPERTIES)) )
{
    // 设定 EAX 听众预设失败
}
```

EAX 提供环境属性的预设, 就是预先定义一组属性用于表示听众的环境。这个特征自从 EAX 1.0 就已提供而且用法简单, 因为不需要太多的编程。然而, 假如你的游戏中有超过一个的环境, 你可能不想使用环境的预设。原因是, 假如你正在使用 EAX 2.0 或者前面的版本, 那就不可能在两个预设的环境之间做渐变效果。一个较好的方法是创建你自己的包含单个属性的预设集, 这样你自己就可能对设定集进行插值。因为你正在自己处理, 你能够在不同次时候插入不同的设定, 例如混响最后被插进 (征询音效设计师的建议)。无论在什么版本下, 你自己执行多个环境的插值以保证它总是工作的。在一些情况下, 比如当一个玩家正在准备下水时, EAX 不应该使用渐变。在这个情况下, 你应该立即转换环境。

为了手动执行环境渐变效果, 你能够像下面那样做 (为了保证可读性), 在这里 t 代表过去渐变时间的百分比。

```
/*
[每一帧]
1) 找到包含玩家的最好声音环境。
2) 假如正在进入或者从水下环境出来, 跳过插值然后立即指定新的环境。
3) 否则, 对当前环境属性线性插值逼近最佳的环境。
*/
// 使用你选择存储这个方法取得当前最好的环境
best_environment = get_sound_environment();
current_environment = &sound_globals.environment;
new_environment->decay_time = t * best_environment->decay_time +
(old_environment->decay_time * (1 - t));
```

6. EAX 音源属性集

EAX 音源属性集构建在 DirectSound 次级缓冲器的对象上, 通过增加环境属性来描述音源环境以及它怎样和听众环境关联。例如, EAX 2.0 和后面版本中, 封闭属性代表根据封闭对象的特征把音源衰减和过滤的总数应用到音源。

每个音源都有自己的属性, 所以最终输出的混合值可能包含很多不同的环境音效。这也意味着你将需要对游戏中每个 3D 音源的声道都存储 EAX 音源属性。为了在一个缓冲器上取得和设定 EAX 音源属性, 必须在硬件中创建带 DSBCAPS_CTRL3D 标记的 DS3D 次级缓冲

器：否则所有的属性集调用都会失败。假如在硬件中创建一个缓冲器用于分配声音的时候失败，但那时还试图为那个缓冲器设定 EAX 默认值，你就可能碰到这种情况。

你需要从每个想控制的 DS3D 缓冲器那获得一个 EAX 音源属性集接口。为了取得属性集接口 (IID_IksPropertySet)，你将需要在每个 3D 次级缓冲器上调用函数 QueryInterface()。

```
LPKSPROPERTYSET pEAXSource[n] = NULL;
if ( SUCCEEDED(hr = p3DBuffer[i]->QueryInterface(
    IID_IksPropertySet, (void**)&pEAXSource[i])) &&
    pEAXSource[i] != NULL )
{
    // 成功获得接口
}
```

7. 查询是否支持音源属性集

就像我们在 EAX 听众对象中做的那样，在使用它们之前，你需要确认用户的声卡是否支持 EAX 音源属性。

```
ULONG support = 0;

if( FAILED(pEAXListener->QuerySupport(
    DSPROPSETID_EAX_BufferProperties,
    DSPROPERTY_EAXBUFFER_ALLPARAMETERS, &support)) )
{
    // 不支持 EAX 听众属性
}

if( (support &
    (KSPROPERTY_SUPPORT_GET|KSPROPERTY_SUPPORT_SET))
    != (KSPROPERTY_SUPPORT_GET|KSPROPERTY_SUPPORT_SET) )
{
    // 不支持 EAX 听众属性
}
```

8. 音源属性集的高级和低级控制

所有版本的 EAX 都提供对音源属性集的高级和低级控制。高级控制就是通过允许 EAX 的混响引擎在内部处理音源属性来降低增加 EAX 到你的系统的复杂性。对于一些游戏来说这可能能接受，但大部分情况你希望支持音源属性，这样声音会适当地被封闭和遮断。低级控制允许你对于每个 3D 音源都设定环境属性。这个设定需要多做一点工作，因为它将需要一个方法来跟踪或者请求每个音源所占据的环境，但是这种的 3D 感受也将更加接近玩家在游戏中看到的那样。

你将需要决定使用哪个 EAX 版本。写这篇文章的时候是 EAX 3.0，也称为 EAX Advanced HD [EAX01]。然而，最流行的版本且能够充分利用硬件性能的是 EAX 2.0 [EAX00]。最通常的选择是努力支持所有的版本，这意味着你有两种选择：自己解释 EAX 版本调用，或者使用 Creative Labs 的 EAX 统一的 DLL 帮助你处理。如果你选择前者，就必须创建单独的

EAX 音效对象，通过它来帮你将引擎中的 EAX 设定（可能存为 EAX 3.0）转换为当前 EAX 版本可能提供的设定。但如果情况是后者，则库将自动转换 EAX3.0 的调用到正确版本的调用（在硬件不支持 EAX 的情况下则什么都不做）。Creative Labs 推荐使用 EAX 统一接口，因为它能免去你要支持每个版本的烦恼，但这个决定还是需要你来做。

为了使用 EAX 统一接口，所有你需要做的就是改变创建 DirectSound 接口对象的方法。你将调用 EAX 版本函数 EAXDirectSoundCreate8，而不再调用 DirectSound 的函数 DirectSoundCreate。下面样例代码给出怎样与 EAX.lib 库进行静态链接。

```
HRESULT hr;
LPDIRECTSOUND8 lpDS8 = NULL; // DirectX 7 应用程序应该使用 LPDIRECTSOUND
// DirectX 7 应用程序应该使用 EAXDirectSoundCreate
if ( FAILED(hr = EAXDirectSoundCreate8(NULL,
    &lpDS8, NULL)) )
{
    printf("EAXDirectSoundCreate8 failed!");
    return hr;
}
```

在通过 EAX 统一接口创建 DirectSound 对象后，你能够像一个标准 DirectSound 对象那样调用 DirectSound 函数。

你应当从 Creative Labs 网站下载和阅读 EAX 统一文档，这可以让你得到进一步的解释。表 7.5.1 列出了在不同版本的 EAX 中可用的听众和音源属性。

表 7.5.1 比较 EAX 不同版本提供的特性

听众 (Listener)	EAX 1.0	EAX 2.0	EAX 3.0	音源 (Source)	EAX 1.0	EAX 2.0	EAX 3.0
预先环境设定 (Environment Preset)	Yes	Yes	Yes	直接通道控制 (Direct)	No	Yes	Yes
音量 (Volume)	Yes	No	No	高频直接通道控制 (Direct HF)	No	Yes	Yes
衰减 (Damping)	Yes	No	No	房间环境 (Room)	No	Yes	Yes
房间环境 (Room)	No	Yes	Yes	房间高频环境 (Room HF)	No	Yes	Yes
房间高频环境 (Room HF)	No	Yes	Yes	房间环境的滚降因子 (Room Roll-off Factor)	No	Yes	Yes
房间低频环境 (Room LF)	No	No	Yes	阻隔 (Obstruction)	No	Yes	Yes
房间环境的滚降因子 (Room Roll-off Factor)	No	Yes	Yes	阻隔低频比率 (Obstruction LF Ratio)	No	Yes	Yes
衰减时间 (Decay Time)	Yes	Yes	Yes	封闭 (Occlusion)	No	Yes	Yes
高频衰减比率 (Decay HF Ratio)	No	Yes	Yes	封闭低频比率 (Occlusion LF Ratio)	No	Yes	Yes

续表

听众 (Listener)	EAX 1.0	EAX 2.0	EAX 3.0	音源 (Source)	EAX 1.0	EAX 2.0	EAX 3.0
低频衰减比率 (Decay LF Ratio)	No	No	Yes	封闭房间比率 (Occlusion Room Ratio)	No	Yes	Yes
反射 (Reflections)	No	Yes	Yes	外面高频音量 (Outside Volume HF)	No	Yes	Yes
反射延迟 (Reflections Delay)	No	Yes	Yes	空气吸收的因子 (Air Absorption Factor)	No	Yes	Yes
反射漂移 (Reflections Pan)	No	No	Yes	互斥 (Exclusion)	No	No	Yes
环境大小 (Environment Size)	No	Yes	Yes	低频互斥比率 (Exclusion LF Ratio)	No	No	Yes
扩散环境 Environment Diffusion 环境	No	Yes	Yes	多普勒因子 (Doppler Factor)	No	No	Yes
空气吸收高频 (Air Absorption HF)	No	Yes	Yes	滚降因子 (Roll-off Factor)	No	No	Yes
混响 (Reverb)	No	No	Yes	调制时间 (Modulation Time)	No	No	Yes
混响延迟 (Reverb Delay)	No	No	Yes	调制深度 (Modulation Depth)	No	No	Yes
混响漂移 (Reverb Pan)	No	No	Yes	高频参考 (HF Reference)	No	No	Yes
回声时间 (Echo Time)	No	No	Yes	低频参考 (LF Reference)	No	No	Yes
回声深度 (Echo Depth)	No	No	Yes				

9. 材质预设定

在计算声音的封闭部分提到材质的预设定问题，可以自己创建或者也可以通过 EAX 创建。假如封闭对象是可以传递声音的，材质的预设定则可以帮助你设定封闭属性或者障碍属性。Creative 对于封闭和障碍属性使用相同的范围，所以仅仅需要创建一个材质来表示封闭和障碍。

一个材质的预设定由 4 个值组成（假如用于封闭，其中两个就无效）。

- Occlusion/Obstruction: 在高频下的衰减。
- Occlusion/Obstruction LF Ratio: 低频。
- Occlusion Room: 房间效果控制。
- Occlusion Direct Ratio: 直接通道控制（通常为 1.0）。

为了应用一个材质预设定，简单使用 EAX 音源属性集和传递用于封闭的标记 DSPROPERTY_EAXBUFFER_OCCLUSIONPARAMETERS 或者用于障碍的标记 DSPROPERTY_

EAXBUFFER_OBSTRUCTIONPARAMETERS 和材质预设定队列。

EAX 3.0 提供预定义 8 个材质预设定：单扇窗、双扇窗、薄门、厚门、木墙、砖墙、石墙和窗帘。这可能还没有完全覆盖你的游戏引擎，你可以很容易地从零开始创建，或者可以使用一个预设定材质为基础进行修改。



在附带的 CD-ROM 上有一个 EAX demo，它包含一个非常简单的环境描述不同的 EAX 环境封闭和障碍效果。

10. ZoomFX

EAX 和 DS3D 遗漏（而 A3D 提供）的一个特征就是表示作为一个 3D 音量的音源。现在，在它们的定位算法中，EAX 和 DS3D 都使用 3D 点音源（point source）。这使得要表现很大或者不规则的音源发射器有点困难。对象比如蟋蟀的鸣叫或者赛车游戏中尘土飞扬的土块是很好用点音源来表示，因为和玩家相比它们的区域很小。（当然，假如玩家在一昆虫大战游戏中扮演一只蚂蚁，则相比较大的蟋蟀应该有它悦耳的脚来代表一区域！）

从距离来看，一点源来表示对象更加实用。然而，当听众解决这个对象，声音发散出的区域占据听众头部四周一大部分区域，同样的方法也充满到听众的视区（field of view）。因此，需要很多的虚拟音源才会产生现实的效果（参见图 7.5.3）。

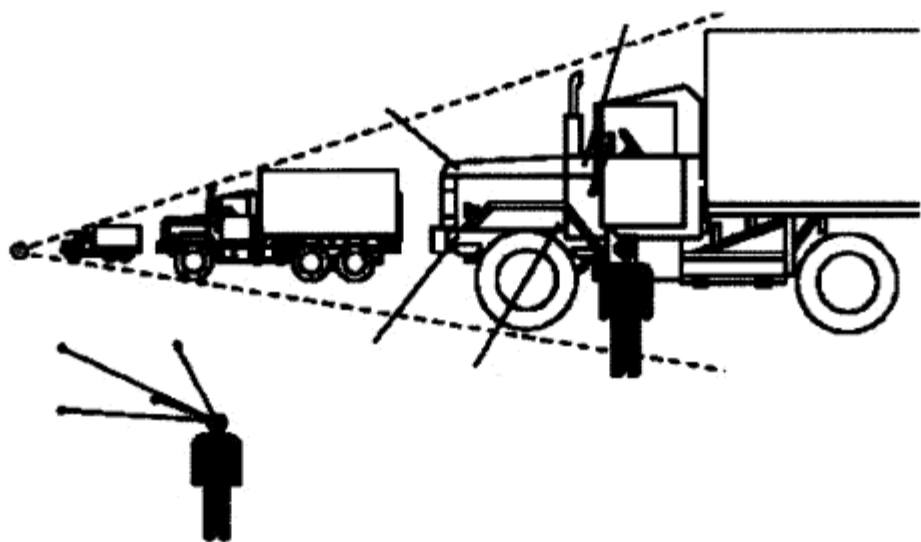


图 7.5.3 ZoomFX 动态抗相关效果

考虑这种情况，在游戏开发的早期，当创建一个音频引擎用于 3D 平台的游戏。我们的 3D 引擎程序员已经刚好完成水的特效，它使得美工可以增加水到关卡中的能力，我们的音频制作人员渴望增加水的环境音效。我们很快意识到一件事情，在表示狭长汹涌的河流可以流过整个关卡，仅仅点音源就需要大量的声音覆盖它的区域。不但我们使用很多的音源，而且我们也被迫在每个音源的最小距离上打主意，以让它们和下一个音源重叠。DS3D 最小和最大距离是使用球形体来测量的，没有必要覆盖整个河流。这使河流音效以完整的音量播放即使当玩家没有很靠近河流（参见图 7.5.4 和图 7.5.5）。

现在进入 Sensuara 公司的 ZoomFX API [Zoom02] 世界。ZoomFX 是开放标准的 API，通过允许音效设计师用一个与声音有关的大小尺寸来表示声音发射对象来弥补鸿沟。类似 EAX，

ZoomFX API 支持音效的硬件通过 DirectSound 属性集来访问。ZoomFX 是一个扩展的音源的缓冲器，它不使用 DS3D 的听众接口。

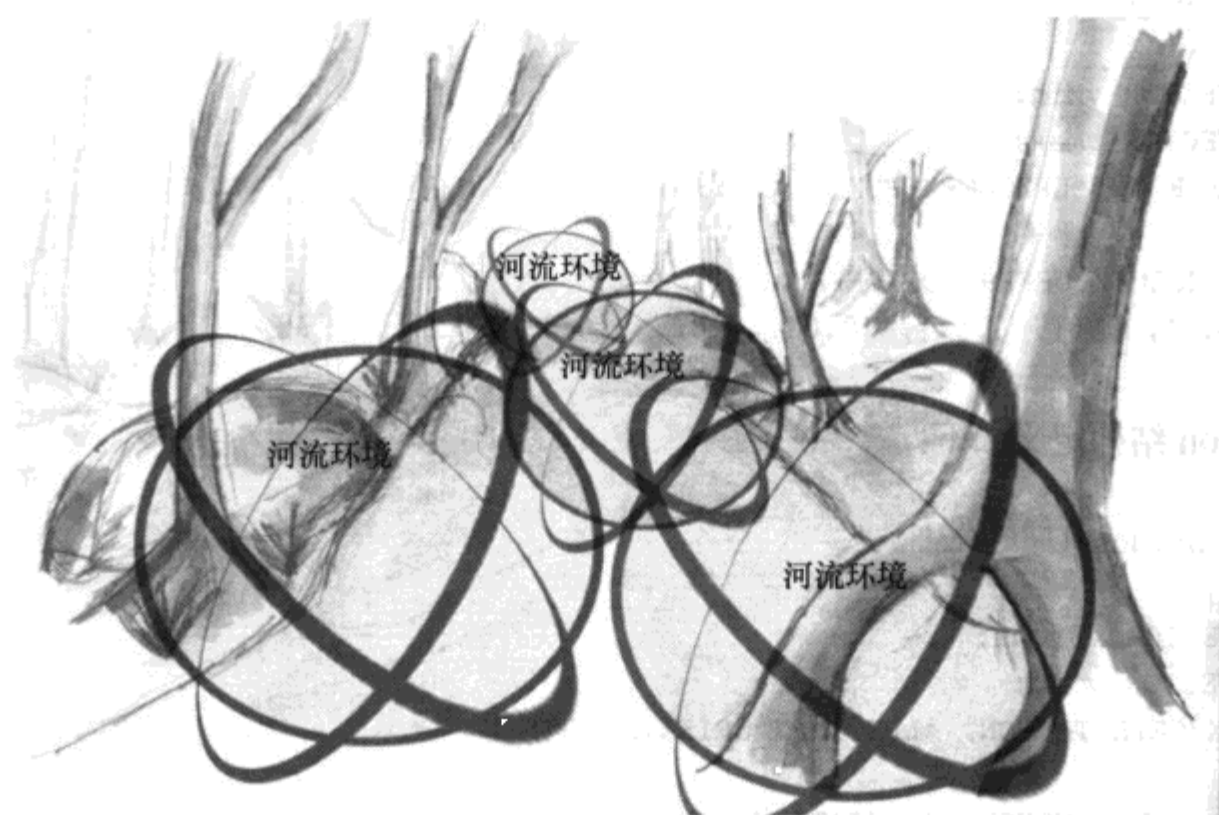


图 7.5.4 通过点音源来表现河流

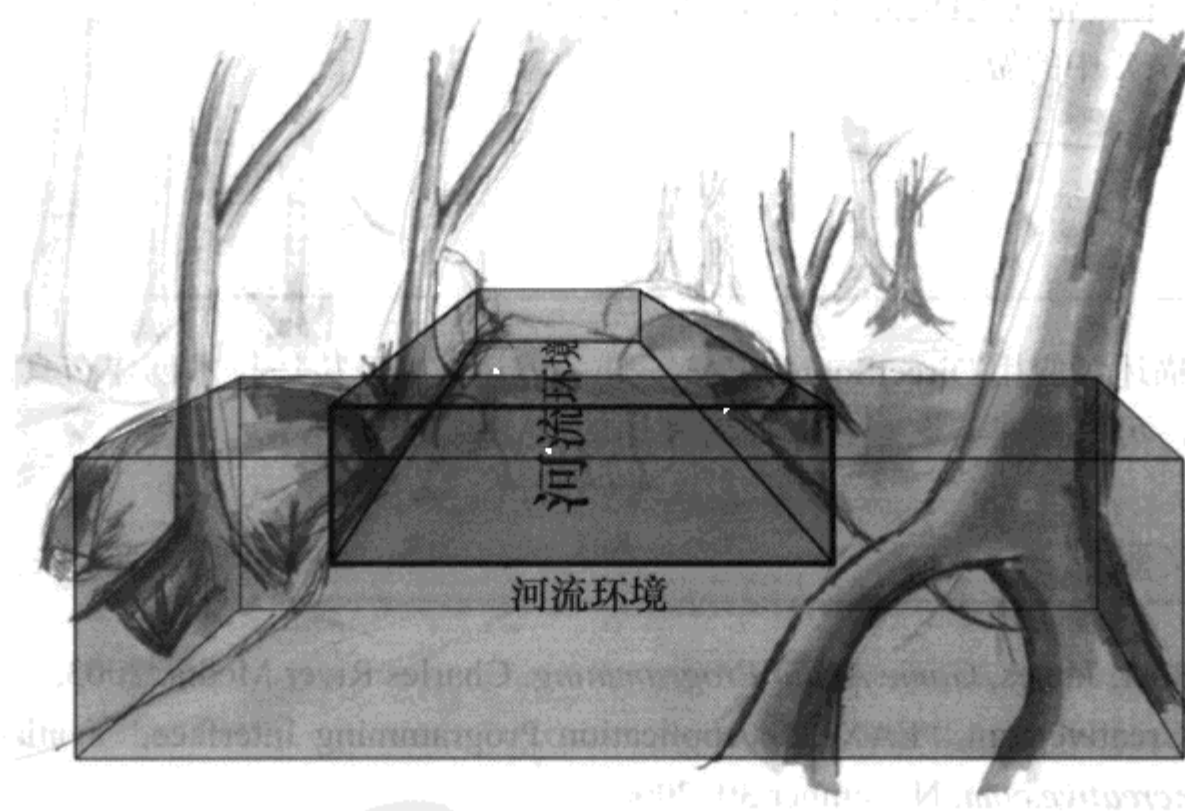


图 7.5.5 通过 ZoomFX 体积来表现河流

把 ZoomFX 加到你的音频引擎是直截了当的。对于每个将接受到 ZoomFX 音效的音源，你将需要存储一个沿坐标轴的包围盒（AABB，Axis-Aligned Bounding Box），一个向前的向量和一个向上的向量。这些属性将传递到 Sensuara 硬件上创建一个巨大的 3D 体积音源。一旦设定代表音源的包围盒，你将可以通过改变向前和向上的向量来随意旋转包围盒。

ZoomFX 的实现非常类似于 EAX；所有的属性都通过 DS3D 属性集方法表示。下面是一个 ZoomFX 使用的包围盒定义的结构体：

```
typedef struct
{
    D3DVECTOR vMin;
    D3DVECTOR vMax;
} ZOOMFX_BOX, *LPZOOMFX_BOX;

#define ZOOMFXBUFFER_BOX_DEFAULT \
    { {0.0f, 0.0f, 0.0f}, {0.0f, 0.0f, 0.0f} }
```

Orientation 结构定义如下：

```
typedef struct
{
    D3DVECTOR vFront;
    D3DVECTOR vTop;
} ZOOMFX_ORIENTATION, *LPZOOMFX_ORIENTATION;

#define ZOOMFXBUFFER_ORIENTATION_DEFAULT \
    { {0.0f, 0.0f, 0.0f}, {0.0f, 0.0f, 0.0f} }
```



在附带的 CD-ROM 上有一个 Demo，示范如何取得和设定属性，这些都非常类似 EAX 中的做法。

7.5.5 总结

这篇文章描述了应用 DirectSound、EAX 和 ZoomFX API 接口技术创建仿真的环境音效。若要深入讨论，可以创建与这些 API 配合使用的 PAS。

7.5.6 参考文献

[Boer03] Boer, James, *Game Audio Programming*. Charles River Media, 2003.

[EAX00] Creative.com, “EAX 2.0 Application Programming Interface,” available online at <http://developer.creative.com>, November 30, 2000.

[EAX01] Creative.com, “EAX 3.0 Application Programming Interface,” available online at <http://developer.creative.com>, November 30, 2001.

[Zoom02] Sibbald, Alastair, “ZoomFX for 3D-sound,” available online at www.sensaura.com/whitepapers/pdfs/devpc012.pdf.

7.6 在游戏的物理引擎中控制实时声音

作者: Frank Luchs, Visionmedia Software Corporation

E-mail: gameprogramminggems@visionmedia.com

译者: 万太平

审校: 许竹钧

在本文中, 我们将描述怎样把音频子系统合成到引擎的物理系统中。如果音频系统和你的物理引擎保持一致, 那你就能避免乏味的手动配音处理过程——空间环境应该总是和游戏中的动画保持完美同步。我们不是提供普通的解决方案, 而是把精力集中于在虚拟世界中驾驶小汽车这个特殊的例子上 (或者其他交通工具)。

我们主要目标是找到一个方法来生成动态和逐步展开的声音, 它可以自动调整参数和选择时间。我们刚好想充分利用已经有的场景, 让游戏物理和几何环境以交互方式驱动音效。使用这个方法, 使用随着时间变换的频谱内容, 对于其他方法很难实现的复杂环境, 能够自动产生精确同步的声音, 特别是有静态音频样本。

这篇文件使用 DirectScene 引擎。这个引擎结合 Sphinx Modular Media, SphinxMM [Luchs02] 系统技术和 Open Dynamics Engine (ODE) [Smith00] 物理引擎。

首先将会看一下我们引擎的主要元素, 然后描述合成方法和对于听得见对象属性的影响, 最后是 Demo 演示。它是受 Brown BuggyDemo [Brown03] 的启发而开发的一个交通工具模拟, 表示在一个逼真的环境中我们模型具体的实现。

7.6.1 游戏引擎

在游戏中用实体 (entity) 表示不连续的对象。一个实体就是我们主要数据结构场景图 (SceneGraph) 中的一个普通节点。是负责管理子系统专门定义的对象属性的容器 (container)。这些是图形对象、物理对象和音频对象, 对应游戏中的图形、物理和音频引擎。图 7.6.1 高度概括了 DirectScene 系统。

例如, 我们场景中每个可能的障碍都分别与一个图形网格 (mesh)、一个碰撞网格和一个碰撞的声音相关联。

虽然对于一面墙壁要渲染的网格可能由上百个多边形组成, 但音频系统仅需要一个缩减多边形的版本。在我们的场景图中, 一个对象对应不同

的 LOD 引用不同的版本。对于大部分引擎做碰撞检测的时候都使用缩减多边形后的几何体；这些简化细节的对象也可以用于音频方面。

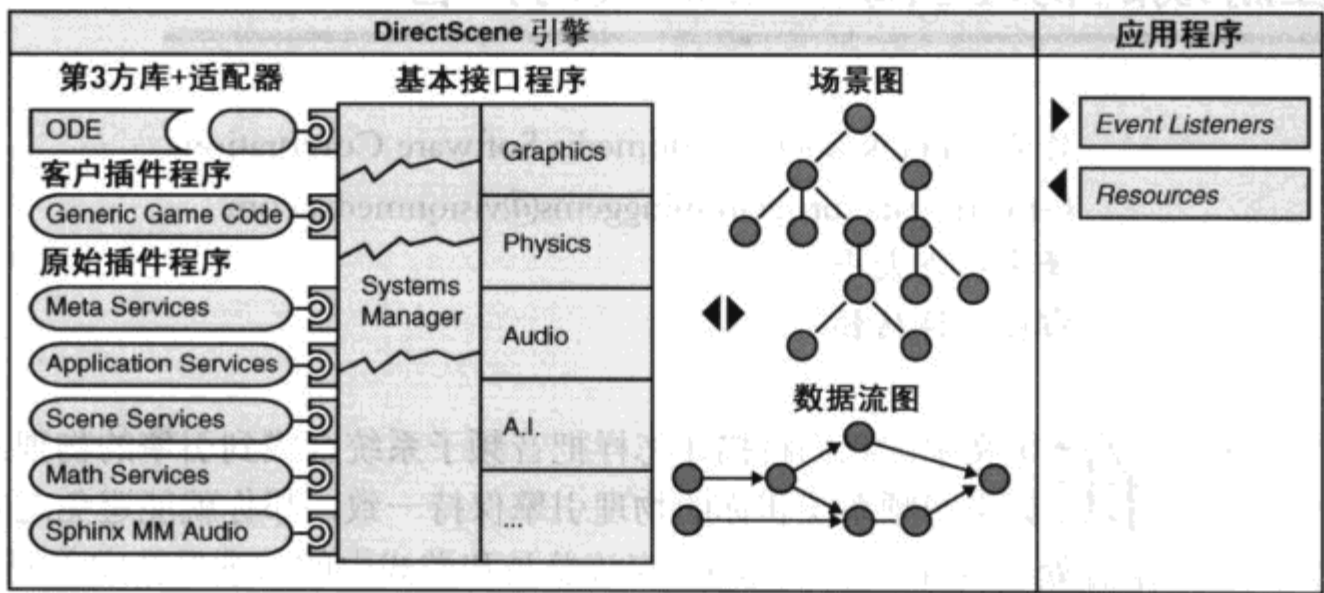


图 7.6.1 引擎概述，包括场景图、图形、物理引擎和音频渲染子系统

引擎通过对象实例化创建动态音源。一些音源通过游戏逻辑预先定义，而其他根据子系统产生的事件来创建。

我们能够使用交通工具模拟的例子来说明。在游戏开始，我们对于背景音乐创建一个音源，它无需进一步的处理和任何空间信息。我们也创建动力引擎的声音作为我们交通工具模拟的部分。负责合成发动机声音的音色，它有调制输入（modulation input），可以通过从物理系统取得的速度和从图形系统取得的位置信息来动态更新。在挤压轮胎或者和另外对象碰撞的情况下，基于接触的时间，我们可以创建和取消音源并根据介入对象的物理状态设定调制参数。

在复杂的环境中，我们有很多可能的接触同时发生。为了效率起见，我们限制并发的音源数量，就是那些与前台事件相关同时活跃的音源。我们也可以为以后使用来预先创建音频的音色，可以避免不必要的和资源消耗大的对象创建和销毁的过程。

1. 音色（Audio Patch）

一种合成方法是通过音色来定义的，描述在单个声音处理单元之间数据流和包含声音产生和调制所有需要的状态。一个典型的音色 Patch，有一个声音发生器顺带一个或者多个处理滤波器。声音发生器播放事先录好的音频样本或者作为音频振荡器基于某一特定的算法来合成音频。描述声音音色的信息类似图形世界中的纹理和材质脚本，在这里，表面属性被映射到特定的对象。声音样本是真实世界的元素，类似在图形系统中使用数字化的图片图像作为纹理。合成的声音是失真的元素，类似程序处理过的纹理，像图形中的火和水特效。为了效率，音色能够被几个实体共享。

2. 同步

虚拟对象的同步处理是通过从单个子系统到应用程序都提供事件听众来完成的。为了达到同步，我们使用一个基于定时器（timer）的循环，允许指定一个准确的帧速率。音频缓冲

器处理是以块状处理,按照 20 微秒顺序有一个典型的缓冲器大小(和因此是延时状态)来匹配 50Hz 的图形渲染帧速率。

7.6.2 混合声音合成

我们特殊的方法是基于通过混合一个随机的噪音成分和一个确定的类似正弦曲线成分的建模声音的一个混合合成。这种方法相对那种明确的物理建模,还提供另外一种选择,允许简单控制从其他子系统直接取得的参数。这些参数或者和共鸣的对象相关,或者和相互交互作用的力相关。

虽然以丢失一些精确性和真实性为代价,但我们没有基于精确的物理原理做振动分析,而是把精力集中在较高计算效率声音渲染方法上。通常,即使会丢失不重要的部分,在声音有关的特定方面做一些夸大是可取的。

我们仅仅实现了两种不同的方法,尽管很简单,我们将获得一个广泛的自然界碰撞声音来模拟对石头、金属或者木头制造的日常对象的音频响应。这两个方法工作原理相反但彼此互补。

我们合成的第一个声音是基于噪音的不规则类型的声音,它是通过重叠和增加多个独立的声音产生的粒子取得的。在[Cook02]中描述了这个 PhISEM (Physically Informed Stochastic Event Modeling) 合成,用于模拟没有音高的打击乐器,如沙球、小手鼓和锯琴。原始噪音数据是通过混合随机振幅和穿过一个单一的两级谐振滤波器指数衰减的相位得到。

这个方法对于那些音质不明显的木头和塑料是很好的,对于连续接触的声音,如刮擦声、滑声和旋转声,它在两个共鸣的形状之间模拟一系列无序的细微碰撞系列。

刮擦噪音常用于我们模拟的声音中;因此,合成算法必须有效率。我们的多粒子噪音是非常简单的模型,对于每个声音仅仅需要两个随机数计算,两个指数方面的衰减和一个引起共鸣的过滤计算。这个方法甚至能够模拟轮胎与地面之间摩擦的声音。有一种很具挑战性的方法,就是根据场景地形纹理构建粒子大小(比如沥青、沙砾)。这对于自动参数化的探索又进了一步。该功能正在开发中。

第二个声音是一个基于阻尼振荡(模式)来合成的。振动模式是一个指定对象潜在频谱类型的振动。我们仅仅取指数衰减和较弱耦合的正弦曲线。衰减声音波动公式的衰减系数依靠对象的属性(质量、大小和材质等等)和碰撞的能量,衰减也是和频率成比例的。一个更高的模式是更强的衰减因子。摩擦系数也是依赖材质的,在对象的表面近似不变。两个模型的参数相对输入的力有线性响应。

对于和弦及其金属组成的谐振对象发生碰撞的高低音,这个方法很好。增加一个整数比率到正弦会导致谐音;使用非整数的关联会导致不和谐的声音。调制输入参数是频率对应振动对象的自然模型,正弦初始化振幅和衰减系数描述在时间域内频谱的变化。

在两个碰撞的主要部分的情况下,音频系统基于指定的声音材质执行下面的步骤。

(1) 我们以不规则的声音开始,作为脚本的音素。由多个粒子合成模仿两个对象在粗糙表面互相摩擦而产生声音。单个的两极滤波器的共鸣调谐到一个适合接触对象音量的值。对于快速重复的脉冲,我们的系统不会对每个新的碰撞都重新设定上冲。这类似合成器上的连奏音:虽然当时仍然按在这个键上但不会重新触发这个音符,而是播放下个音符。

- (2) 假如碰撞能量足够高可以造成波形谐振 (shape resonance), 我们增加基于正弦的信号。
- (3) 假如我们有一个指定的样本, 它将按照噪音包络线的波峰被触发。

7.6.3 可听见对象的属性

动态声音控制是直截了当的, 假如我们访问场景图中各种对象的属性。我们能够获得物理和几何特性, 比如大小、弹性、质量和形状, 然后指定它们的属性到音频控制器。这些控制器负责在值之间映射处理过程。它们会做所有我们需要的计算: 单元化、缩放、限定范围、加等等。

在我们的声音环境中, 有两种典型的对象: 静止的网格和动态的网格。静止的网格用于场景地形和不动的元素, 而动态的网格用于对象的动画。在大多数情况, 对于静止的元素, 单独的多粒子噪音就足够了, 而动态的网格受到更多物理系统的影响, 需要单独的音频属性来平衡视觉效果和期望的声音。

7.6.4 对象形状的影响

一个对象的形状决定特有的频谱。让我们看看一些基本几何体, 了解一下要创建可靠的声音需要设定什么模型。

- 一段简单绳子仅仅有调和比 (harmonic ratio)。
- 一条刚性的棒有这些模型: 1.0、2.765、5.404 和 8.933。例如一条金属棒子按照一稀疏的非谐波的频谱发出鸣叫, 以较高模式衰减得很快。
- 矩形的膜有非常密集的声音频谱, 会发出丰富且复杂的声音。
- 圆形的膜有谐振 (特征函数 eigenfunctions), 是贝塞耳函数 (Bessel functions)。
- 环形的盘子相对圆形的膜来说没有那么密集的频谱。
- 打开的或者封闭的管子有谐波的模型; 一根一边封闭管子有不成对的多个基本频率。
- 盒子共鸣器有幅频响应, 是梳状谐波的叠加, 每个都有一个基本频率, 按照如下计算:

$$f_{0,l,m,n} = \frac{c}{2} \sqrt{(l/x)^2 + (m/y)^2 + (n/z)^2} \quad (7.6.1)$$

公式中 c 是声音的速度; l, m, n 是没有任何普通公因子的三元整型组; x, y 和 z 是盒子的边长。

球形谐振器有频率响应, 是不规则梳状波的叠加而产生的, 每个有峰在球形贝塞耳函数的最尽头的点:

$$fns = \frac{c}{2\pi r} * zns \quad (7.6.2)$$

fns 是频率响应, c 声音的速度, r 是球体的半径, zns 是 n 阶贝塞耳函数 s 阶导数。

对于前面的公式更加详细的信息, 可以参考 [Avanzini01]、[O'Brien02] 和 [Riegel00]。

以下规则可以帮助我们实时调整合成。

- 产生的声音依靠受影响对象的冲撞位置。对于接近模型边缘的碰撞点, 较高频率会变得相对更加兴奋。换句话说, 声调在接近中心时变得“闷”而在边缘时变得“亮”。

- 对于强大的力量，则非线性效果开始活动。
- 对于单个分音，一个刚刚好的衰减率相比谐振材质中心音高起的作用更加大 [Avanzini01]。
- 长期接触使得振动模型高频，它的表面短于本身接触的时间。
- 接触时间 t_0 （例如，在对象从另外一个对象移开的时间）有一个主要目的定义最初调子频谱的特性。较短的 t_0 对应类似脉冲的瞬态有丰富的频谱，因此提供一个明快的上冲；类似一个较长的 t_0 对于一个较为平滑的瞬态在高频区域的能量很少。

7.6.5 对象材质的影响

材质的声音特性能够使用内在摩擦力的系数来表现。这个参数测量弹性的度数和定义振动衰减时间和频谱分量的带宽。当我们通过内在的系数对材质归类，我可以得到两组：一组包含塑料和木头的柔软材质，以及一组类似玻璃和金属的坚硬材质。在从塑料到金属频谱我们有日渐增加衰减次数和初步减少得带宽 [Avanzini01]。这是对于随意振动的形状是对的；衰减的玻璃类似塑料。

7.6.6 撞击和碰撞

在原理上，对于一个单一的冲撞和多个连续的接触，我们不会使用不同的合成方法来处理，因为他们有一个平滑的转变。象弹跳、刮擦、旋转和打碎这样复杂的声音，能够在不同的时间比例上重复简单的碰撞来模拟。

大部分碰撞都有一个“刹车 (skid)”部分，表面在碰撞过程中以很短的时间从各自上滑过。例如弹跳的球将比旋转的球更快刹车。在频繁接触的情况下，声音混合会扩大噪音部分。

主要的问题是对创建和销毁一个接触过程的跟踪，当它是充分的开始和调制相关的音色 Patch。

我们的物理系统 ODE 需要我们在处理碰撞时候提供 `NearCallback()` 函数。系统提供一系列“接触接合点 (contact joints),”它是对象对在指定时刻碰撞。这个很不幸，因为没有很便利的来做单个声音的触发，因为这个列不但包含已经发生的碰撞，也包括简单彼此静止的对象，相对持续逗留的情况。大部分的这些接触不会产生声音，所以我们应该禁止它们；否则，我们听到的恐怕是类似机枪的声音！一种方法是调整这个比较当前帧的碰撞列和以前帧。当一对物体第一次接触，我们触发一碰撞声音。

一个相关问题是，不会以永久的接触停留，对象可能“弹起来”和频繁碰撞。在这种情况下，我们不想每次重新触发声音。特别假如声音是快速“上冲”阶段，我们有能够以“机枪”的噪音结束。有一种方法可以促成更加自然的声音就是控制声音振幅，特别是在上冲阶段，根据接触的频率。我们随着一段时间合剂脉冲，然后逐步增加已经播放声音的等级。

7.6.7 演示

在 Demo 中，你驾驶一辆车子通过一个有不同障碍物的环境。和一个障碍物碰撞会产生

合适的声音。引擎声音通过车辆的速度控制。另外的效果是轮胎/地面的噪音，根据地面类型不同而变化，和轮胎的轧轧声。一个有机器的礼堂显示闭塞/音频入口 (Portal) 和预先计算的混响概念。这个 Demo 可以在 www.directscene.com 上找到。

1. 音源

通过指定速度向量来很直率的计算我们车辆的速度。在 2D 情况我们计算 $\sqrt{v_x^2 + v_z^2}$ 。对于简单的模型，我们将仅仅映射这个值到音源音高调制 (pitch modulation)。对于复杂的模型，我们将控制一个合成引擎的声音，将通过输入速度参数以一个更加复杂的方法来改变。

2. 翻滚效果 (Rolling)

对于有不同粗粗度的地面纹理阐述了使用单独过滤噪音部分的不同摇摆声音效果。一些地面包含或多或少的规则样式，不能很好的使用我们过滤部分噪音来模拟。表面这样的不规则通过增加声音振幅调制到脚本噪音上面。

3. 滑动效果 (Slipping)

在我们的例子中，我们也模拟“轧轧声 (squeaking)”当车轮在地面滑行（例如因为刹车）。为了一个自动的轧轧声我们比较没有滑行的值和当前车轮的角速度。我们知道一个纯滚动的车轮没有任何滑动有如下的角速度

$$\omega = \frac{v}{2\pi r}$$

r 是车轮的半径， v 是径向速度。

我们使用两个绝对不同的速度来控制轧轧声发生器的级别。（当然，我们仅仅使用这个声音当车轮与地面接触的时候！）

7.6.8 总结

直接从物理模型实时渲染真实对象的声音将变得越来越重要。DirectScene graph 实现从 3D 图形 API 明白的范例和基于知道的几何和物理信息创建听觉方面的环境细节。手动声音创建和同步是复杂和花费时间的过程，通过使用音频控制器用于实时地监听物理和几何属性然后映射它们到有用的合成参数。

基于结构的音色 Patch 能够很容易的扩展描述新的合成方法用于环境音效。

7.6.9 参考文献

[Avanzini01] Avanzini, Frederico, and Davide Rocchesso, “Controlling Material Properties in Physical Models of Sounding Objects,” available online at www.soundobject.org/papers/avaroc_icmc2001.pdf, 2001.

[Brown03] Brown, Si, “BuggyDemo,” available online at <http://freefall.freehosting.net/downloads/buggydemo.html>, 2002.

[Cook02] Cook, Perry R., *Real Sound Synthesis for Interactive Applications*, AK Peters LTD, 2002.

[Luchs02] Luchs, Frank, "Real-Time Modular Audio," *Game Programming Gems 3*, Charles River Media, 2002.

[O'Brien02] O'Brien, James F., Chen Shen, and Christine M. Gatchalian. "Synthesizing Sounds from Rigid-Body Simulations," available online at <http://citeseer.nj.nec.com/518076.html>, 2002.

[Riegel00] Edward Riegelsberger, Micah Mason, and Suneil Mishra, "Advancing 3D Audio through an Acoustic Geometry Interface," available online at www.gdconf.com/archives/2000/riegelsb.pdf, 2000.

[Smith2000] Smith, Russell, "Open Dynamics Engine," available online at <http://opende.sourceforge.net>, 2000.



关于光盘

随书光盘中包含与文章有关的全部源程序。有对应的源程序的文章共有如下这些：

- 1.2: 一个基于 HTML 的日志和调试系统
- 1.3: 时钟: 游戏的脉搏尽在掌握
- 1.5: 利用模版化的空闲块列表克服内存碎片问题
- 1.6: 一个用 C++ 实现的泛型树容器类
- 1.7: 弱引用和空对象
- 1.8: 游戏中的实体管理系统
- 1.9: Windows 和 Xbox 平台上地址空间受控的动态数组
- 1.10: 用临界阻尼实现慢入慢出的平滑
- 1.11: 一个易用的对象管理器
- 1.12: 使用自定义的 RTTI 属性对对象进行流操作及编辑
- 1.13: 使用 XML 而不牺牲速度
- 2.1: 使用马其赛特旋转的 Zobrist 散列法
- 2.2: 抽取截锥体和 camera 信息
- 2.3: 解决大型游戏世界坐标中的精度问题
- 2.4: 非均匀样条
- 2.5: 用协方差矩阵计算更贴切的包围盒
- 2.6: 应用于反向运动的雅可比转置方法
- 3.1: 死神的十指: 战斗中的命中算法
- 3.3: 编写基于 Verlet 积分方程的物理引擎
- 3.4: 刚体动力学中的约束器
- 3.5: 在动力学模拟中的快速接触消除法
- 3.6: 互动水面
- 3.7: 用多层物理模拟快速变形
- 3.8: 快速且稳定的形变之模态分析
- 4.3: 非玩家角色决策: 处理随机问题
- 5.2: 非封闭网络模型的 GPU 容积阴影构架
- 5.3: 透视阴影贴图
- 5.4: 结合使用深度和基于 ID 的阴影缓冲
- 5.7: 实时半调色法: 快速而简单的样式化阴影
- 5.8: 在 3D 模型中应用团队色的各种技术

- 5.10: 使用场景亮度采样实现动态的 Gamma
- 5.15: 使用地平线进行地形遮挡剔除
- 6.2: 支持成千上万个客户端的服务器
- 6.4: 在客户/服务器环境下运用并行状态机
- 6.5: 位打包: 一种网络压缩技术
- 7.3: 动态变量和音频编程
- 7.4: 创建一个音频脚本系统
- 7.5: 使用 EAX 和 ZoomFX API 用于环境音频解决方案

光盘中还包含了连接代码所需的一些第三方函数库。

有关本书的更多信息, 包括勘误表和更新等, 可以在 www.GameProgrammingGems.com 网站上找到。

系统需求

Windows: Intel® Pentium®或 AMD Athlon 系列, 推荐使用较新型的处理器。需要 Windows 98 (64MB 内存) 或 Windows 2000 (128MB 内存) 或其较新版本。推荐使用 3D 图形加速卡以求最优性能。须安装 DirectX 9、GLUT 3.7 或其较新版本。

Linux: Intel® Pentium®或 AMD Athlon 系列, 推荐使用较新型的处理器。需要 2.4.x 或较新版本的 Linux 内核。推荐使用 32MB 内存。推荐使用 3D 图形加速卡以求最优性能。须安装 XFree86 4.0、GLUT 3.7、OpenGL 驱动程序、glibc 2.1 或其较新版本。也可以用 Mesa 取代 3D 硬件支持。

